第一章. 介绍

本章简要地介绍了用户自定义函数(UDF)及其在 Fluent 中的用法。在 1.1 到 1.6 节中我们会介绍一下什么是 UDF；如何使用 UDF，以及为什么要使用 UDF，在 1.7 中将一步步的演示一个 UDF 例子。

1.1　什么是 UDF？
1.2　为什么要使用 UDF？
1.3　UDF 的局限
1.4　Fluent5 到 Fluent6 UDF 的变化
1.5　UDF 基础
1.6　解释和编译 UDF 的比较
1.7 一个 step-by-stepUDF 例子

1.1 什么是 UDF？

用户自定义函数，或 UDF，是用户自编的程序，它可以动态的连接到 Fluent 求解器上来提高求解器性能。用户自定义函数用 C 语言编写。使用 DEFINE 宏来定义。UDF 中可使用标准 C 语言的库函数，也可使用 Fluent Inc.提供的预定义宏，通过这些预定义宏，可以获得 Fluent 求解器得到的数据。

UDF 使用时可以被当作解释函数或编译函数。解释函数在运行时读入并解释。而编译 UDF 则在编译时被嵌入共享库中并与 Fluent 连接。解释 UDF 用起来简单，但是有源代码和速度方面的限制不足。编译 UDF 执行起来较快，也没有源代码限制，但设置和使用较为麻烦。

1.2 为什么要使用 UDF？

一般说来，任何一种软件都不可能满足每一个人的要求，FLUENT 也一样，其标准界面及功能并不能满足每个用户的需要。UDF 正是为解决这种问题而来，使用它我们可以编写 FLUENT 代码来满足不同用户的特殊需要。当然，FLUENT 的 UDF 并不是什么问题都可以解决的，在下面的章节中我们就会具体介绍一下 FLUENT UDF 的具体功能。现在先简要介绍一下 UDF 的一些功能：

VISIT…

LANZAROTE Caliente.COM

- 定制边界条件，定义材料属性，定义表面和体积反应率，定义 FLUENT 输运方程中的源项，用户自定义标量输运方程（UDS）中的源项扩散率函数等等。
- 在每次迭代的基础上调节计算值
- 方案的初始化
- （需要时）UDF 的异步执行
- 后处理功能的改善
- FLUENT 模型的改进（例如离散项模型，多项混合物模型，离散发射辐射模型）

由上可以看出 FLUENT UDF 并不涉及到各种算法的改善，这不能不说是一个遗憾。当然为了源代码的保密我们还是可以理解这样的做法的。其实，如果这些代码能够部分开放，哪怕就一点点，我想 FLUENT 会像 LINUX 一样发展更为迅速，使用更为广泛。遗憾的是，从目前来看，这只是一种幻想。什么时候中国人可以出自己的精品？

1.3 UDF 的局限

尽管 UDF 在 FLUENT 中有着广泛的用途，但是并非所有的情况都可以使用 UDF。UDF 并不能访问所有的变量和 FLUENT 模型。例如，它不能调节比热值；调节该值需要使用求解器的其它功能。如果您不知道是否可以用 UDF 解决某个特定的问题，您可以求助您的技术支持。

1.4 Fluent5 到 Fluent6UDF 的变化

如果你有 FLUENT5 的 UDF 编程经验，请注意在 FLUENT6 种的下列变化：

- FLUENT6 中加入了大量的通用多相模型。When one of these general multiphase models is enabled, storage must be set aside for the mixture as well as the individual phases. This functionality is manifested in the code through the use of additional thread and domain data structures. Consequently, some predefined macros have been added that allow access to data contained within mixture-level and phase-level domain and thread structures. See Section 3.11 for details on writing UDF for multiphase applications.

  If you have a **FLUENT** 5 UDF with an external domain declaration that you want to use in **FLUENT** 6, then the extern statement must be replaced by a call to the Get_Domain utility and assignment to a Domain pointer as shown below. The Fluent-provided utility, Get_Domain(1), returns the pointer to the mixture-level domain. See Section 6.5.1 for more details on Get_Domain.

  Example

```
extern Domain *domain;

DEFINE_ON_DEMAND(my_udf)
{
...
}
```

is to be replaced by

```
DEFINE_ON_DEMAND(my_udf)
{
Domain *domain;
domain = Get_Domain(1);
...
}
```

The macro C_VOF accesses volume fraction values from the **FLUENT** solver. C_VOF(c, pt[i]) has two arguments, c and pt[i]. c is the cell identifier. pt[i] is the pointer to the corresponding phase-level thread for the *i*th phase, where i is the phase_domain_index. For example, C_VOF(c,pt[i]) can be used to return the volume fraction of the *i*th phase fluid at cell c. The pointer pt[i] can also be retrieved using THREAD_SUB_THREAD, discussed in Section 6.5.4, using i as an argument.

- For compiled UDF, the makefile called `Makefile.udf` that was provided in previous **FLUENT** releases has been renamed to `makefile.udf2`. See Section 7.3.2 for more details.
- For multiphase flow problems, you will need to supply your own user-defined scalar flux function instead of using the default function provided by **FLUENT**.
- `DEFINE_PROPERTY` is to be used to define UDF for particle or droplet diameter for the mixture model, previously the Algebraic Slip Mixture Model (ASMM), instead of the `DEFINE_DRIFT_DIAM` macro.

1.5 UDF 基础

- 1.5.1 输运方程
- 1.5.2 单元（Cells）,面，区域（Zones）和线（Threads）
- 1.5.3 操作
- 1.5.4 求解器数据
- 1.5.5 运行

1.5.1 输运方程

FLUENT 求解器建立在有限容积法的基础上，这种方法将计算域离散为有限数目的控制体或是单元。网格单元是 **FLUENT** 中基本的计算单元，这些单元的守恒特性必须保证。也就是说普通输运方程，例如质量，动量，能量方程的积分形式可以应用到每个单元：

$$\underbrace{\frac{\partial}{\partial t}\int_V \rho\phi dV}_{\text{unsteady}} + \underbrace{\oint_A \rho\phi V \cdot dA}_{\text{convection}} = \underbrace{\oint_A \Gamma\nabla\phi \cdot dA}_{\text{diffusion}} + \underbrace{\int_V S_\phi dV}_{\text{generation}} \tag{1.5.1}$$

此处，ϕ 是描述普通输运数量的变量（a general transportable quantity）,根据所求解的输运方程它可取不同的值。下面是在输运方程中可求解的 ϕ 的子集。

Transport Equation	**Variable for** ϕ
continuity	1
x momentum	velocity (*u*)
y momentum	velocity (*v*)
z momentum	velocity (*w*)
energy	enthalpy (*h*)
turbulent kinetic energy	*k*
turbulent dissipation rate	ϵ
species transport	mass fraction of species (Y_i)

守恒与否需要知道通过单元边界的通量。因此，需计算出单元和面上的属性值（properties）。

1.5.2 单元（Cells）,面，区域（Zones）和线（Threads）

单元和单元面被组合为一些区域（zones），这些区域规定了计算域（例如，入口，出口，壁面）的物理组成（physical components）。当用户使用 **FLUENT** 中的 UDF 时，用户的 UDF 可调用流体区域或是边界区域的计算变量（solution variables）。UDF 需要获得适当的变量，比如说是区域参考（a zone reference）和单元 ID，以便标定各个单元。

区域（A zone）是一群单元或单元面的集合，它可以由模型和区域的物理特征（比如入口，出口，壁面，流体区域）来标定。例如，一些被指定为面域（a face zone）的单元面可以被指定为 velocity-inlet 类型，由此，速度也就可指定了。线（A thread）是 **FLUENT** 数据结构的内部名称，可被用来指定一个区域。Thread 结构可作数据储存器来使用，这些数据对于它所表示的单元和面来说是公用的（The Thread structure acts as a container for data that is common to the group of cells or faces that it represents）。

1.5.3 操作

多数的 UDF 任务需要在一个线的所有单元和面上重复执行。比如，定义一个自定义轮廓函数（a custom profile function）则会对一个面线上（in a face thread）的所有单元和面进行循环。 为了用户方便， Fluent Inc.向用户提供了一些循环宏工具（looping macro utilities）来执行对单元，面，节点（nodes）和线（threads）的重复操作。例如，单元循环宏（Cell-looping macros ）可以对给定单元线上的所有单元进行循环操作（loop over cells in a given cell thread allowing access to all of the cells）。 而面循环宏（Face-looping macros）则可调用所有给定面线（a given face thread）的面。Fluent 提供的循环工具请见 Chapter 6。

在某些情况下，UDF 需要对某个变量操作，而这个变量恰恰又不能直接被当作变量来传递调用。比如，如果用户使用 DEFINE_ADJUST 宏来定义 UDF，求解器将不会向它传递 thread 指针。这种情况下，用户函数需要用 Fluent 提供的宏来调用线指针（thread pointer）。见 Chapter 6。

1.5.4 求解器数据

通过 **FLUENT** 用户界面将 C 函数（它已被编译和连接）连接到求解器上可实现调用求解器变量。一旦 UDF 和求解器正确连接，无论何时，函数都可调用求解器数据。这些数据将会被作为用户变量自动地传递给 UDF。注意，所有的求解器变量，不管是求解器传递给 UDF 的，还是 UDF 传递给求解器的，都使用 SI 单位。

1.5.5 运行

UDF 将会在预定时刻被 FLUENT 调用 。但是，也可对它们进行异步执行，使用 DEFINE_ON_DEMAND 宏，还可在需要时（on demand）执行。详情请见 4.2.3

1.6 解释和编译 UDF 的比较

编译 UDF 和 **FLUENT** 的构建方式一样。脚本 Makefile 被用来调用 C 编译器来构建一个当地目标代码库（a native object code library）。目标代码库包含高级 C 语言源代码的机器语言翻译。代码库在 **FLUENT** 运行时由“动态加载”（``dynamic loading"）过程连接到 **FLUENT** 上。连接后，与共享库的联系（the association with the shared library）将会被保存在用户的 case 文件中,这样,当 **FLUENT** 以后再读入 case 文件时,此编译库将会与 **FLUENT** 自动连接。这些库是针对计算机的体系结构和一定版本的 **FLUENT** 使用的。所以，当 **FLUENT** 更新，或计算机操作系统改变，或是在不同类型的机器上运行时，这些库必须重新构建。

而解释 UDF 则是在运行时，直接从 C 语言源代码编译和装载（compiled and loaded directly from the C source code）。在 **FLUENT** 运行中，源代码被编译为中介的独立于物理结构的使用 C 预处理程序的机器代码（an intermediate, architecture-independent machine code）。当 UDF 被调用时，机器代码由内部仿真器（an internal emulator），或注释器（interpreter）执行。注释器不具备标准 C 编译器的所有功能；它不支持 C 语言的某些原理（elements）。所以，在使用 interpreted UDF 时，有语言限制（见 3.2）。例如，interpreted UDF 不能够通过废弃结构（dereferencing structures）来获得 **FLUENT** 数据。要获得数据结构，必须使用由 **FLUENT** 提供的预定义宏。另一个例子是 **FLUENT** interpreter 不能识别指针数组。这些功能必须由 compiled UDF 来执行。

编译后，用户的 C 函数名称和内容将会被储存在 case 文件中。函数将会在读入 case 文件时被自动编译。独立于物理结构的代码的外层（This extra layer of architecture-independent code）可能会导致执行错误（a performance penalty），但却可使 UDF 共享不同的物理结构，操作系统，和 Fluent 版本。如果运行速度较慢，UDF 不用被调节就可以编译代码的形式（in compiled mode）运行。**FLUENT** 中的 compiled 和 interpreted UDF 请见 Chapter 7。

选择 interpreted UDF 或是 compiled UDF 时，注意以下内容：

- Interpreted UDF
  - 对其它平台是便捷的（portable）。
  - 可作为（compiled UDF）来运行。
  - 不需 C 编译器。
  - 比 compiled UDF 慢 。
  - 需要较多的代码。
  - 在使用 C 语言上有限制。
  - 不能与编译系统或用户库（compiled system or user libraries）连接。
  - 只能使用预定义宏来获得 FLUENT 结构中的数据。（见 Chapters 5 和 6）。
- Compiled UDF
  - 比 interpreted UDF 运行快。
  - 在使用 C 语言上不存在限制。
  - 可用任何 ANSI-compliant C 编译器编译。
  - 能调用以其他语言编写的函数 (specifics are system- and compiler-dependent)。
  - 机器物理结构需要用户建立 **FLUENT** (2D or 3D) 的每个版本的共享库（a shared library for each version of **FLUENT** (2D or 3D) needed for your machine architecture）。
  - 如果包含有注释器（interpreter）不能处理得 C 语言元素，则不能作为（interpreted UDF ）运行。

总的来说，当决定使用那种类型的 UDF 时：

- 使用 interpreted UDF 作为简单的函数
- 使用 compiled UDF 作为复杂的函数，这些函数
  - 对 CPU 有较大要求（例如每次运行时，在每个单元上均须调用的属性 UDF（a property UDF）。
  - 需要使用编译库（require access to a compiled library）。

1.7 一个 step-by-stepUDF 例子

编辑 UDF 代码，并且在用户的 **FLUENT** 模型中有效使用它，须遵循以下七个基本步骤：

1. 定义用户模型。
2. 编制 C 语言源代码。
3. 运行 **FLUENT** ，读入，并设置 case 文件。
4. 编译或注释（Compile or interpret）C 语言源代码。

5. 在 **FLUENT** 中激活 UDF。
6. 开始计算。
7. 分析计算结果，并与期望值比较。

在开始解决问题前，用户必须使用 UDF 定义希望解决的问题（Step 1）。例如，加入用户希望使用 UDF 来定义一个用户化的边界条件（a customized boundary profile ）。用户首先需要定义一系列数学方程来描述这个条件。

接下来用户需要将这些数学方程（概念设计，conceptual design）用 C 语言写成一个函数（Step 2）。用户可用文本编辑器来完成这一步。以.c 为后缀名来把这个文件保存在工作路径下。

写完 C 语言函数后，用户即可运行 FLUENT 并且读入或设置 case 文件（Step 3）。对 C 语言源代码进行注释，编译，和调试（interpret, compile, and debug），并在 FLUENT 中激活用户函数 （Step 5）。最后， 运行计算（Step 6），分析结果并与期望值比较。（Step 7）。根据用户对结果的分析，可将上述整个过程重复几次。具体如下。

Step 1: 定义用户模型

生成和使用 UDF 的第一步是定义用户的模型方程。

如图 Figure1.7.1 所示的涡轮叶片。模拟叶片周围的流场使用了非结构化网格。计算域由底端的周期性边界（ a periodic boundary on the bottom ）延伸到顶端的相同部分（an identical one on the top），速度入口在左边，压力出口在右边。

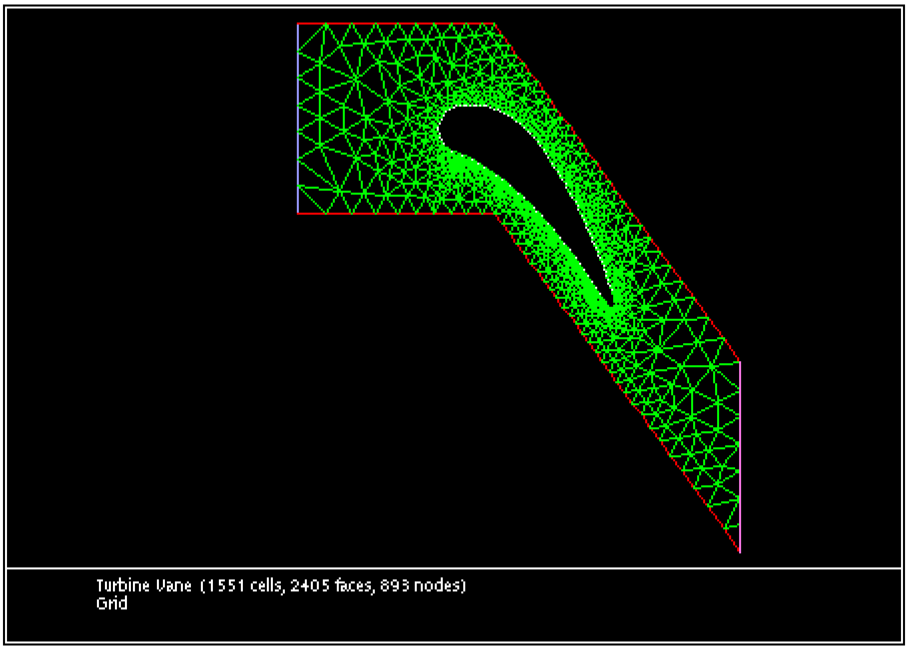


Figure 1.7.1: The Grid for the Turbine Vane Example

文中对入口 *x* 速度为常数分布和抛物线分布的流场进行了比较。分段线性的分布可由边界场选项得到（the application of a profile using a piecewise-linear profile is available with the boundary profiles option），而多项式分布则只能使用用户自定义函数得到。

进口速度为常数（20 m/s）的结果如图1．7．2和1．7．3所示。当流动沿着涡轮叶片进行时，初始速度场被改变了。

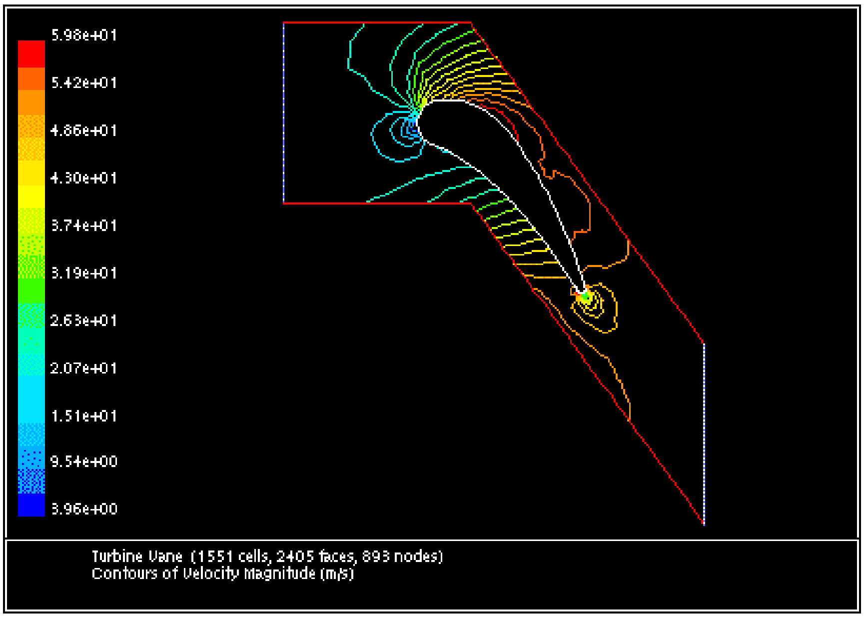


Figure 1.7.2: Velocity Magnitude Contours for a Constant Inlet *x* Velocity

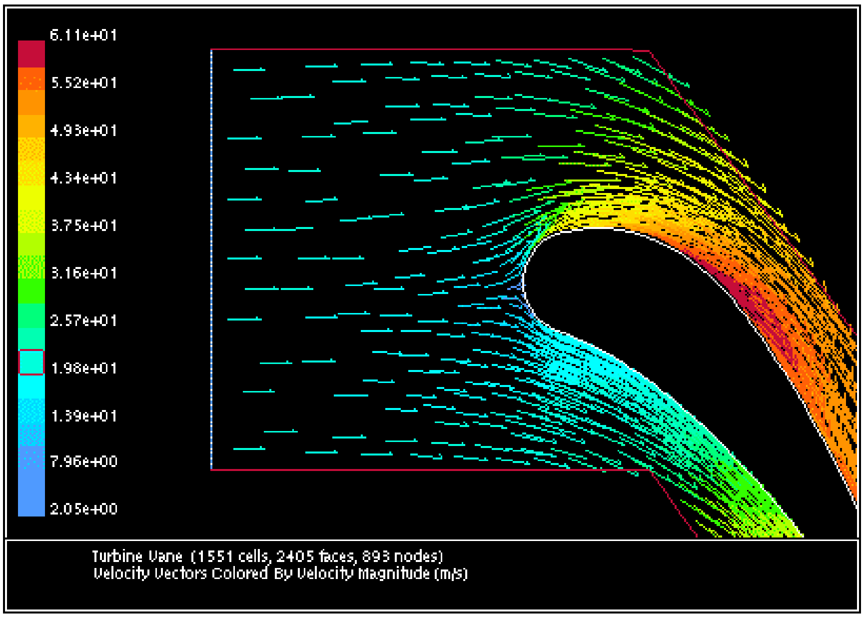


Figure 1.7.3: Velocity Vectors for a Constant Inlet *x* Velocity

假定现在要设涡轮叶片入口速度 *x* 不是一常数值，其分布如下

$$v_x = 20 - 20\left(\frac{y}{0.0745}\right)^2$$

变量 y 在入口中心处为 0． 0，在入口上部和下部则分别为 ± 0.0745 m 而入口中心处的 x 速度为 20 m/s ，边界上为 0 。

用户可用 UDF 描述这一分布，并将它应运到 **FLUENT 模型中**来解决这类问题。

Step 2: 编制 C 语言源代码。

选定方程定义 UDF 后，用户可用任意文本编辑器来书写 C 语言代码。以扩展名.c 保存源代码文件保存到工作路径下。关于 UDF 的书写请参考 Chapter 3 。

下面是一个怎样在 UDF 中应用方程的例子。UDF 的功能由主要的 DEFINE 宏（the leading DEFINE macro）来定义。此处，DEFINE_PROFILE 宏用来表示下面的代码旨在给求解器提供边界的轮廓信息。书中将在以后部分讨论其它的 DEFINE 宏。

```
/***********************************************************************/
/* udfexample.c                                                        */
/* UDF for specifying a steady-state velocity profile boundary condition */
/***********************************************************************/
#include "udf.h"
DEFINE_PROFILE(inlet_x_velocity, thread, index)
{
real x[ND_ND];                /* this will hold the position vector */
real y;
face_t f;
begin_f_loop(f, thread)
{
F_CENTROID(x,f,thread);
y = x[1];
F_PROFILE(f, thread, index) = 20. - y*y/(.0745*.0745)*20.;
}
end_f_loop(f, thread)
}
```

DEFINE_PROFILE 宏的第一个变量 inlet_x_velocity 用来定义速度入口面板中的函数。名称可任意指定。在给定的边界区域上的所有单元面（identified by f in the face loop）上将会使用函数的这个方程。当用户在 **FLUENT** 用户界面选定 UDF 作为边界条件时，将会自动定义线（thread）。下标由 begin_f_loop **应用程序**自动定义。UDF 中，begin_f_loop 被用来形成对边界区域上所有单元面的循环（loop through all cell faces in the boundary zone）。对于每个面，面的质心（the face centroid）的坐标可由 F_CENTROID 宏来获得。抛物线方程中用

到了 *y* 坐标 y ，速度值通过 F_PROFILE 宏来返回给面。 begin_f_loop 宏和 F_PROFILE 宏都是 **FLUENT** 提供的宏。详情请见 Chapter 5 。

Step 3: 运行 **FLUENT** ，读入，并设置 case 文件

建立 UDF 后，用户开始设置 FLUENT。

1. 在工作路径下启动 **FLUENT**。

2. 读入（或设置）case 文件（如果 case 文件以前设置过，请确认它是否被保存在了工作路径下）。

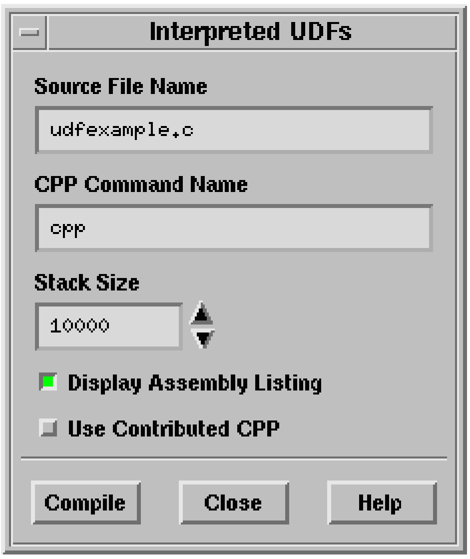
Step 4: 编译或注释（Compile or interpret）C 语言源代码

这部分将例中的源代码作为 interpreted UDF 来编译。注意，这个例子不可应用于 Windows 的并行网络（Windows parallel networks）。完整的编译和连接 UDF 请见 Chapter 7 。

1. 确认 UDF 的 case 文件（如果以前设置过）和 C 语言源代码在工作路径下。
2. 用 **Interpreted UDF** 面板编译 UDF (例如， `udfexample.c`) 。

Define → **User-Defined** → **Functions** → **Interpreted...**

Figure 1.7.4: The **Interpreted UDF** Panel



(a) 在 **Source File Name** 下键入 C 语言源代码文件 (例如， `udfexample.c`) 。

!! 如果用户源代码不在目前工作路径下，则在编译 UDF 时，需在 Interpreted UDF 面板中间如文件完整的路径。

(b) 在 **CPP Command Name** 一栏里，选择 C 预处理器。
(c) **Stack Size** 缺省设置为 10000。如果用户函数的局部变量数目大于 10000，将会导致堆栈溢出。这种情况下，应将 Stack Size 设置为比局部变量大的数。

(d) 点击 **Compile** 编译 UDF。

存储 case 文件时，C 语言代码的名称和内容将会储存在 case 文件中。如果 Interpreted UDF 面板中的 Display Assembly Listing 选项被选中，当编译进行时，控制台窗口中将会显示汇编语言代码。另外，此选项将会被存储在 case 文件中，当以后用户在执行 FLUENT 任务时，控制台窗口中将会编译时一样显示汇编语言代码。

```
inlet\_x\_velocity:
.local.pointer thread (r0)
.local.int nv (r1)
0              .local.end
0              save
.local.int f (r3)
1              push.int 0
.local.pointer x (r4)
3              begin.data 8 bytes, 0 bytes initialized:
7              save
.                .
.                .
156              pre.inc.int f (r3)
158              pop.int
159              b .L3 (22)
.L2:
161              restore
162              restore
163              ret.v
```

!! 注意，如果编译失败，**FLUENT** 将会给出错误信息，请调试程序。详见 7.2.3。

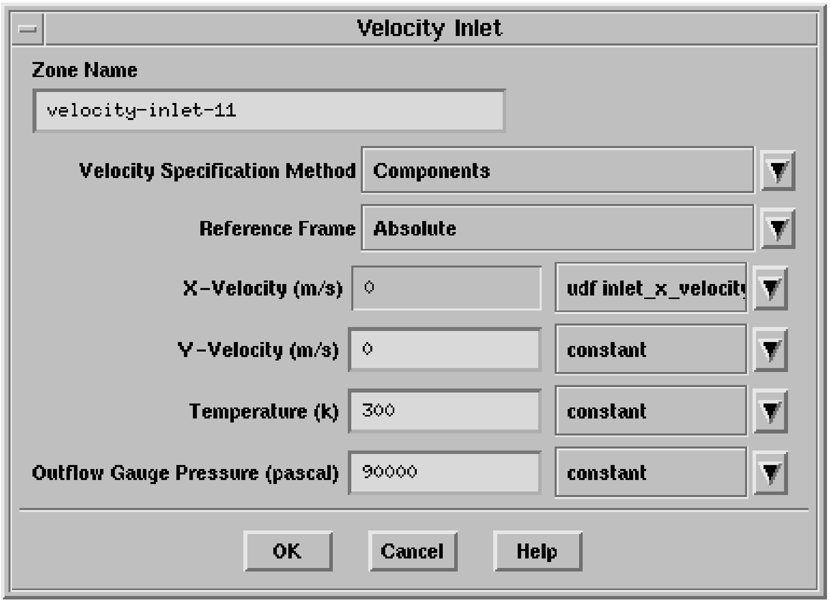
(e) 编译结束后，点击 **Close**。

!! 此例中的 UDF 源代码也可作为 compiled UDF 运行。

Step 5: 在 **FLUENT** 中激活 UDF

编译连接完 UDF 后，FLUENT 用户界面面板中将会看到 UDF。此例中，可在 **Velocity Inlet** 面板中选择 UDF。

Define → **Boundary Conditions...**



在 X-Velocity 下拉列表中，选择 udf inlet_x_velocity，此名称是由例中的函数给定的。一旦选中，UDF 将会替代 X-Velocity 中的 0 值进行运算。点击 OK 接受新的边界条件，关闭面板。

Step 6: 开始计算

运算方式和以前一样。

Solve →Iterate...

Step 7: 分析计算结果，并与期望值比较

计算收敛后，获得一个修正的速度场。Figure1.7.5 为入口 *x* 速度为抛物线分布的速度等值线，可与（Figure 1.7.2）所示的入口速度为常数 20 m/sec 的流场比较。常数条件下，流动在涡轮叶片周围变形（distorted）。入口抛物线分布，在入口中心处为最大值，边缘上为 0。

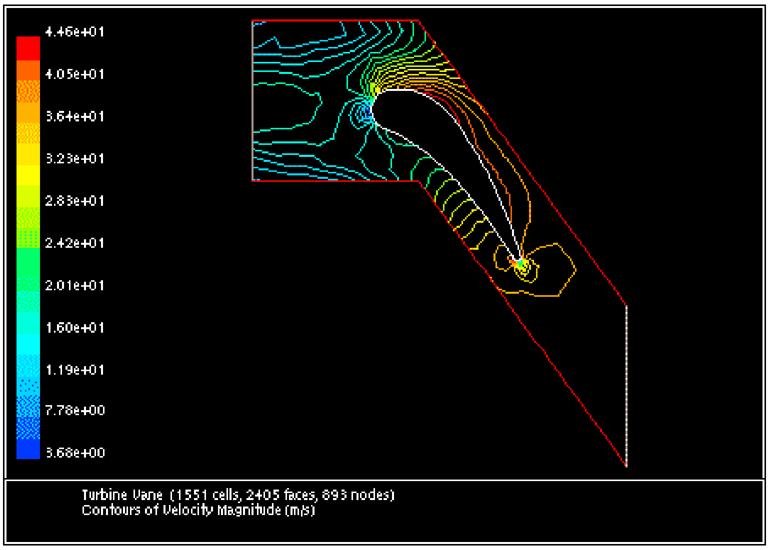


Figure 1.7.5: Velocity Magnitude Contours for a Parabolic Inlet *x* Velocity

第二章.UDF 的 C 语言基础

本章介绍了 UDF 的 C 语言基础

2.1 引言
2.2 注释你的 C 代码
2.3FLUENT 中的 C 数据类型
2.4 常数
2.5 变量
2.6 自定义数据类型
2.7 强制转换
2.8 函数
2.9 数组
2.10 指针
2.11 声明
2.12 常用 C 操作符
2.13C 库函数
2.14 用#define 实现宏置换
2.15 用#include 实现文件包含
2.16 与 FORTRAN 比较

2.1 引言

本章介绍了 C 语言的一些基本信息，这些信息对处理 FLUENT 的 UDF 很有帮助。本章首先假定你有一些编程经验而不是 C 语言的初级介绍。本章不会介绍诸如 while-do 循环，联合，递归，结构以及读写文件的基础知识。如果你对 C 语言不熟悉可以参阅 C 语言的相关书籍。

2.2 注释你的 C 代码

熟悉 C 语言的人都知道，注释在编写程序和调试程序等处理中是很重要的。注释的每一行以“/*”开始，后面的是注释的文本行，然后是“*/”结尾

如：/* This is how I put a comment in my C program */

2.3FLUENT 的 C 数据类型

FLUENT 的 UDF 解释程序支持下面的 C 数据类型：

Int：整型

Long：长整型

Real：实数

Float：浮点型

Double：双精度

Char：字符型

注意：UDF 解释函数在单精度算法中定义 real 类型为 float 型，在双精度算法宏定义 real 为 double 型。因为解释函数自动作如此分配，所以使用在 UDF 中声明所有的 float 和 double 数据变量时使用 real 数据类型是很好的编程习惯。

2.4 常数

常数是表达式中所使用的绝对值，在 C 程序中用语句#define 来定义。最简单的常数是十进制整数（如：0，1，2）包含小数点或者包含字母 e 的十进制数被看成浮点常数。按惯例，常数的声明一般都使用大写字母。例如，你可以设定区域的 ID 或者定义 YMIN 和 YMAX 如下：#define WALL_ID 5

#define YMIN 0.0

#define YMAX 0.4064

2.5 变量

变量或者对象保存在可以存储数值的内存中。每一个变量都有类型、名字和值。变量在使用之前必须在 C 程序中声明。这样，计算机才会提前知道应该如何分配给相应变量的存储类型。

2.5.1 声明变量

变量声明的结构如下：首先是数据类型，然后是具有相应类型的一个或多个变量的名字。变量声明时可以给定初值，最后面用分号结尾。变量名的头字母必须是 C 所允许的合法字符，变量名字中可以有字母，数字和下划线。需要注意的是，在 C 程序中，字母是区分大小写的。下面是变量声明的例子：

```
int n;                          /*声明变量 n 为整型*/
int i1, i2;                     /*声明变量 i1 和 i2 为整型*/
float tmax = 0.;                /* tmax 为浮点型实数，初值为 0 */
real average_temp = 0.0;        /* average_temp 为实数，赋初值为 0.1*/
```

2.5.2 局部变量

局部变量只用于单一的函数中。当函数调用时，就被创建了，函数返回之后，这个变量就不存在了，局部变量在函数内部（大括号内）声明。在下面的例子中，mu_lam 和 temp 是局部变量。

```
DEFINE_PROPERTY(cell_viscosity, cell, thread)
{
  real mu_lam;
  real temp = C_T(cell, thread);

  if (temp > 288.)
    mu_lam = 5.5e-3;
  else if (temp > 286.)
    mu_lam = 143.2135 - 0.49725 * temp;
  else
    mu_lam = 1.;
```

```
  return mu_lam;
}
```

2.5.3 全局变量

全局变量在你的 UDF 源文件中是对所有的函数都起作用的。（调用一个 UDF 源文件可能会包括一系列的连接函数。）它们是在单一函数的外部定义的。全局变量一般是在预处理程序之后的文件开始处声明。

2.5.4 外部变量

如果全局变量在某一源代码文件中声明，但是另一个源代码的某一文件需要用到它，那么你必须在另一个文件中声明它是外部变量。外部变量的声明很简单，你只需要在变量声明的最前面加上 extern 即可。如果有几个文件涉及到该变量，最方便的处理方法就是在头文件（.h）中加上 extern 的定义，然后在所有的.c 文件中引用该头文件即可。只有一个.c 文件应该包括没有 extern 关键字的变量声明，如下所示。注意：extern 只用于编译过的 UDF。

例子：

```
/* filea.h    */
/*包含外部定义的头文件*/
extern real volume;
/* filea.c    */
/*调用头文件 filea.h 中声明的 volumn 的 C 函数*/
#include "udf.h"
#include "filea.h"
real volume;
DEFINE_ADJUST(compute_volume, domain)
{
    /*计算某些区域 volumn 的代码*/
    volume = ....
}
/* fileb.c    */
/*调用头文件 filea.h 中声明的 volumn 的另一个 C 函数*/
#include "udf.h"
#include "filea.h"
DEFINE_SOURCE(heat_source,c,t,ds,eqn)
{
    /*  用总数来计算每个单位体积的源项的代码*/
    /*fliea.c 的 compute_volum 计算出的 volume*/
    real total_source = ...;
    real source;
    source = total_source/volume;
```

```
    return source;
}
```

2.5.5 静态变量

static 声明对于全局变量和局部变量的影响是不一样的。静态局部变量在函数调用返回之后，该变量不会被破坏。静态全局变量则在定义该变量的.c 源文件之外对任何函数保持不可见。静态声明也可以用于函数，使该函数只对定义它的.c 源文件保持可见。下面是静态全局变量声明的例子。注意：extern 只用于编译过的 UDF。

例子：

```
#include "udf.h"
static real abs_coeff = 1.0;      /*  吸收系数*/
real source;
DEFINE_SOURCE(energy_source, c, t, dS, eqn)
{
  int P1 = ....;
  dS[eqn] = -16.* abs_coeff * SIGMA_SBC * pow(C_T(c,t),3.);
  source =-abs_coeff *(4.* SIGMA_SBC * pow(C_T(c,t),4.) - C_UDSI(c,t,P1));
  return source;
}
DEFINE_SOURCE(p1_source, c, t, dS, eqn)
{
  int P1 = ...;
  dS[eqn] = -abs_coeff;
  source = abs_coeff *(4.* SIGMA_SBC * pow(C_T(c,t),4.) - C_UDSI(c,t,P1));
  return source;
}
```

2.6 自定义数据类型

C 还允许你用结构和 typedef 创建自定义数据类型。下面是一个结构列表的定义。注意：typedef 只用于编译过的 UDF。

例子：

```
typedef struct list
{
        int a;
        real b;
        int c;
}
mylist;                          /* mylist 为类型结构列表*/
mylist x,y,z;                    /* x,y,z 为类型结构列表*/
```

2.7 强制转换

你可以通过强制转换将某一数据类型转换为另一种。强制由类型来表示，其中的类型包括 int，float 等等，如下例所示：

```
int x = 1;
real y = 3.14159;
int z = x+((int) y);            /* z = 4 */
```

2.8 函数

函数是用完成一定任务的一系列语句。在定义该函数的同一源代码中，这些任务可能对其它的函数有用，也可能会被用于完成源文件以外的函数中。每个函数都包括一个函数名以及函数名之后的零行或多行语句，其中有大括号括起来的函数主体可以完成所需要的任务。函数可以返回特定类型的数值。C 函数通过数值来传递数据。

函数有很多数据类型，如 real，void 等，其相应的返回值就是该数据类型，如果函数的类型是 void 就没有任何返回值。要确定定义 UDF 时所使用的 DEFINE 宏的数据类型你可以参阅 udf.h 文件中关于宏的#define 声明一节，也可以参阅附录 A 的列表。

!! C 函数不能改变它们的声明，但是可以改变这些声明所指向的变量。

2.9 数组

数组的定义格式为：名字[数组元素个数]，C 数组的下标是从零开始的。变量的数组可以具有不同的数据类型。

例子

```
int a[10], b[10][10];
real radii[5];

a[0] = 1;                        /*  变量 a 为一个一维数组*/
radii[4] = 3.14159265;           /*变量 radii 为一个一维数组*/
b[10][10] = 4;                   /*变量 b 为一个二维数组*/
```

2.10 指针

指针变量的数值是其它变量存储于内存中的地址值。C 程序中指针变量的声明必须以*开头。指针广泛用于提取结构中存储的数据，以及在多个函数中通过数据的地址传送数据。

例如：int *ip;

本语句声明了一个指向整型变量的指针变量 ip。此时你可以为指针变量分配一个地址值了。现在假定你要将某一地址分配给指针 ip，你可以用取地址符&来实现。例如：

ip = &a;

就分配给指针 ip 变量 a 的地址值了。

要得到指针变量所指向的单元的值，你可以使用：

```
*ip
```

你还可以为指针 ip 所指向的变量赋值，例如：

```
*ip = 4;
```

将 4 赋给指针 ip 所指向的变量。下面是使用指针的例子：

```
int a = 1;
int *ip;
ip = &a;                    /* &a 返回了变量 a 的地址值*/
printf("content of address pointed to by ip = %d\n", *ip);
*ip = 4;                    /* a = 4    */
printf("now a = %d\n", a);
```

在上面的语句中，整型变量赋初值为 1。然后为整型变量声明一个指针。然后整型变量 a 的地址值分配给指针 ip。然后用*ip 来输出指针 ip 所指向的值（该值为 1）。然后用*ip 间接的给变量 a 赋值为 4。然后输出 a 的新值。指针还可以指向数组的起始地址，在 C 中指针和数组具有紧密的联系。

2.10.1 作为函数自变量的指针

C 函数可以通过指针进入和修改它们的自变量。在 **FLUENT** 中，线程和域指针是 UDF 常用的自变量。当你在 UDF 中指定这些自变量时， **FLUENT** 解算器会自动将指针所指向的数据传送给 UDF，从而使你的函数可以存取解算器的数据（你不必声明作为自变量从解算器传送给 UDF 的指针）。例如，某一传送给指定（由 DEFINE_PROFILE 宏来定义的）自定义轮廓 UDF 的自变量是一个指向应用于边界条件的线程的指针。DEFINE_PROFILE 函数会存取线程指针所指向的数据。

2.11 控制语句

你可以使用控制语句，如 if, if-else 和循环来控制 C 程序中语句的执行顺序。控制语句决定了程序序列中下一步该执行的内容

2.11.1 if 语句

if 语句是条件控制语句的一种。格式为：

```
if (逻辑表达式)
```

```
    {语句}
```

其中逻辑表达式是判断条件，语句是条件满足时所要执行的代码行。

例子

```
if (q != 1)
    {a = 0; b = 1;}
```

2.11.2 if-else 语句

if-else 语句是另一种条件控制语句。格式为：

```
if (逻辑表达式)
  {语句}
else
  {语句}
```

如果逻辑表达式条件满足，则执行第一个语句，否则执行下面的语句。

例子

```
if (x < 0.)
  y = x/50.;
else
  {
   x = -x;
   y = x/25.;
  }
```

下面是等价的 FORTRAN 代码，大家可以比较一下：

```
IF (X.LT.0.) THEN
   Y = X/50.
ELSE
   X = -X
   Y = X/25.
ENDIF
```

2.11.3 for 循环

for 循环是 C 程序最为基本的循环控制语句。它和 FORTRAN 中的 do 循环很类似。格式为：

```
for (起点;终点;增量)
        {语句}
```

其中起点是在循环开始时执行的表达式；终点是判断循环是否结束的逻辑表达式；增量是循环迭代一次之后执行的表达式（通常是增量计数器）。

例子：

```
/*  输出整数 1-10 及它们的平方*/
int i, j, n = 10;
for (i = 1 ; i <= n ; i++)
        {
                j = i*i;
                printf("%d %d\n",i,j);
        }
```

下面是等价的 FORTRAN 代码，大家可以做一比较：

```
      INTEGER I,J
      N = 10
      DO I = 1,10
      J = I*I
      WRITE (*,*) I,J
      ENDDO
```

2.12 常用的 C 运算符

运算符是内部的 C 函数，当它们对具体数值运算时会得到一个结果。常用的 C 运算符是算术运算符和逻辑运算符。

2.12.1 算术运算符

下面是一些常用的算术运算符。

=　赋值
+　加
-　减
*　乘
/　除
%　取模
++　增量

-- 减量

注意：乘、除和取模运算的优先级要高于加、减运算。除法只取结果的整数部分。取模只取结果的余数部分。++运算符是增量操作的速记符。

2.12.2 逻辑运算符

下面是一些逻辑运算符。

< 小于
<= 小于或等于
> 大于
>= 大于或等于
== 等于
!= 不等于

2.13 C 库函数

当你书写 UDF 代码时，你可以使用 C 编译器中包括的标准数学库和 I/O 函数库。下面各节介绍了标准 C 库函数。标准 C 库函数可以在各种头文件中找到（如：global.h）。这些文件都被包含在 udf.h 文件中。

2.13.1 三角函数

下面的三角函数都是计算变量 x（只有一个还计算 y）的三角函数值。函数和变量都是双精度实数变量。具体的意义大家应该都很清楚，就不具体介绍了。

double acos (double x);	返回 x 的反余弦函数
double asin (double x);	返回 x 的反正弦函数
double atan (double x);	返回 x 的反正切函数
double atan2 (double x, double y);	返回 *x/y* 的反正切函数
double cos (double x);	返回 x 的余弦函数
double sin (double x);	返回 x 的正弦函数
double tan (double x);	返回 x 的正切函数
double cosh (double x);	返回 x 的双曲余弦函数
double sinh (double x);	返回 x 的双曲正弦函数

double tanh (double x);	返回 x 的双曲正切函数

2.13.2 各种数学函数

下面列表中，左边是 C 函数，右边是对应数学函数：

double sqrt (double x);	$\sqrt{x}$
double pow(double x, double y);	x^y
double exp (double x);	e^x
double log (double x);	$\ln(x)$
double log10 (double x);	$\log_{10}(x)$
double fabs (double x);	$\lvert x \rvert$
double ceil (double x);	不小于 x 的最小整数
double floor (double x);	不大于 x 的最大整数

2.13.3 标准 I/O 函数

C 中有大量的标准输入输出（I/O）函数。在很多情况下，这些函数在指定的文件中工作。下面是一些例子。

FILE *fopen(char *filename, char *type);	打开一个文件
int fclose(FILE *fd);	关闭一个文件
int fprintf(FILE *fd, char *format, ...);	格式化输出到一个文件
int printf(char *format, ...);	输出到屏幕
int fscanf(FILE *fd, char *format, ...);	格式化读入一个文件

函数 fopen 和 fclose 分别打开和关闭一个文件。函数 fprintf 以指定的格式写入文件，函数 fscanf 以相同的方式从某一文件中将数据读入。函数 printf 是一般的输出函数。fd 是一个文件指针，它所指向的是包含所要打开文件的信息的 C 结构。除了 fopen 之外所有的函数都声明为整数，这是因为该函数所返回的整数会告诉我们这个文件操作命令是否成功执行。

在下面的例子中，需要打开的数据文件的名字用双引号括起来。fopen 中的选项 r 表明该文件是以可读形式打开的。fscan 函数从 fd 所指向的文件中读入两个浮点数并将它们存储为 f1 和 f2。关于 C 的标准输入输出函数其它更多的信息，你可以查阅相关手册（如：[2]）。

例子：

```
FILE *fd;
    fd = fopen("data.txt","r");      /*    opens a file named data.txt    */
    fscanf(fd, "%f ,%f", &f1, &f2);
    fclose(fd);
```

2.14 用#define 实现宏置换

UDF 解释程序支持宏置换的 C 预处理程序命令。当你使用#define 宏置换命令，C 预处理程序（如，cpp）执行了一个简单的置换，并用替换文本替换宏中定义的每一个自变量。

#define *macro replacement-text*

如下面的宏置换命令：

#define RAD 1.2345

预处理程序会在 UDF 中所有的变量 RAD 出现的地方将 RAD 替换为 1.2345。在你的函数中可能会有很多涉及到变量 RAD 的地方，但是你只需要在宏命令中定义一次，预处理程序会在所有的代码中执行替换操作。

在下面这个例子中：

#define AREA_RECTANGLE(X,Y) ((X)*(Y))

你的 UDF 中所有的 AREA_RECTANGLE(X,Y)都会被替换为(X)和(Y)的乘积。

2.15 用#include 实现文件包含

UDF 解释程序还支持文件包含的 C 前处理命令。当你使用#include 包含一个文件时，C 前处理程序会将#include *filename* 行替换为文件名对应的文件内容。

#include " *filename* "

文件名对应的文件必须在当前目录中。只有 udf.h 文件例外，这是因为 **FLUENT** 解算器会自动将它读入。

如下面的文件包含命令：

#include "udf.h"

会将文件 udf.h 包含进你的源代码中。

2.16 与 FORTRAN 的比较

很多简单的 C 函数和 FORTRAN 函数的子程序很相似，例子如下：

简单的 C 函数	等价的 FORTRAN 函数
int myfunction(int x)	INTEGER FUNCTION MYFUNCTION(X)
{	
int x,y,z;	INTEGER X,Y,Z
y = 11;	Y = 11
z = x+y;	Z = X+Y
printf("z = %d",z);	WRITE (*,100) Z
return z;	MYFUNCTION = Z
}	END

UDF 第 3 章 写 UDF

本章主要概述了如何在 FLUENT 写 UDF。

3.1 概述

3.2 写解释式 UDF 的限制

3.3 FLUENT 中 UDF 求解过程的顺序

3.4 FLUENT 网格拓扑

3.5 FLUENT 数据类型

3.6 使用 DEFINE Macros 定义你的 UDF

3.7 在你的 UDF 源文件中包含 udf.h 文件

3.8 定义你的函数中的变量

3.9 函数体

3.10 UDF 任务

3.11 为多相流应用写 UDF

3.12 在并行中使用你的 UDF

3.1 概述（Introduction）

UDF 是用来增强 FLUENT 代码的标准功能的，在写 UDF 之前，我们要明确以下几个基本的要求。首先，必须用 C 语言编写 UDF。必须使用 FLUENT 提供的 DEFINE 宏来定义 UDF。UDF 必须含有包含于源代码开始指示的 udf.h 文件；它允许为 DEFINE macros 和包含在编译过程的其它 FLUENT 提供的函数定义。UDF 只使用预先确定的宏和函数从 FLUENT 求解器访问数据。通过 UDF 传递到求解器的任何值或从求解器返回到 UDF 的值，都指定为国际（SI）单位。

总之，当写 UDF 时，你必须记住下面的 FLUENT 要求。UDF：

1. 采用 C 语言编写。
2. 必须为 udf.h 文件有一个包含声明。
3. 使用 Fluent.Inc 提供的 DEFINE macros 来定义。
4. 使用 Fluent.Inc 提供的预定义宏和函数来访问 FLUENT 求解器数据。
5. 必须使返回到 FLUENT 求解器的所有值指定为国际单位。

3.2 写解释式 UDF 的限制（Restriction on Writing Interpreted UDF）

无论 UDF 在 FLUENT 中以解释还是编译方式执行，用户定义 C 函数（说明在 Section 3.1 中）的基本要求是相同的，但还是有一些影响解释式 UDF 的重大编程限制。FLUENT 解释程序不支持所有的 C 语言编程原理。解释式 UDF 不能包含以下 C 语言编程原理的任何一个：

1. goto 语句。
2. 非 ANSI-C 原型语法
3. 直接的数据结构查询（direct data structure references）
4. 局部结构的声明
5. 联合(unions)
6. 指向函数的指针（pointers to functions）
7. 函数数组。

在访问 FLUENT 求解器数据的方式上解释式 UDF 也有限制。解释式 UDF 不能直接访问存储在 FLUENT 结构中的数据。它们只能通过使用 Fluent 提供的宏间接地访问这些数据。另一方面，编译式 UDF 没有任何 C 编程语言或其它注意的求解器数据结构的限制。

3.3 FLUENT 求解过程中 UDF 的先后顺序（Sequencing of UDF in the FLUENT Solution Process）

当你开始写 UDF 代码的过程时（依赖于你写的 UDF 的类型），理解 FLUENT 求解过程中 UDF 调用的内容或许是重要的。求解器中包含连接你写的用户定义函数的 call-outs。知道 FLUENT 求解过程中迭代之内函数调用的先后顺序能帮助你在给定的任意时间内确定那些数据是当前的和有效的。

分离式求解器

在分离式求解器求解过程中（Figure 3.3.1），用户定义的初始化函数（使用 DEFINE_INIT 定义的）在迭代循环开始之前执行。然后迭代循环开始执行用户定义的调整函数（使用 DEFINE_ADJUST 定义的）。接着，求解守恒方程，顺序是从动量方程和后来的压力修正方

程到与特定计算相关的附加标量方程。守恒方程之后，属性被更新（包含用户定义属性）。这样，如果你的模型涉及到气体定律，这时，密度将随更新的温度（和压力 and/or 物质质量分数）而被更新。进行收敛或者附加要求的迭代的检查，循环或者继续或停止。

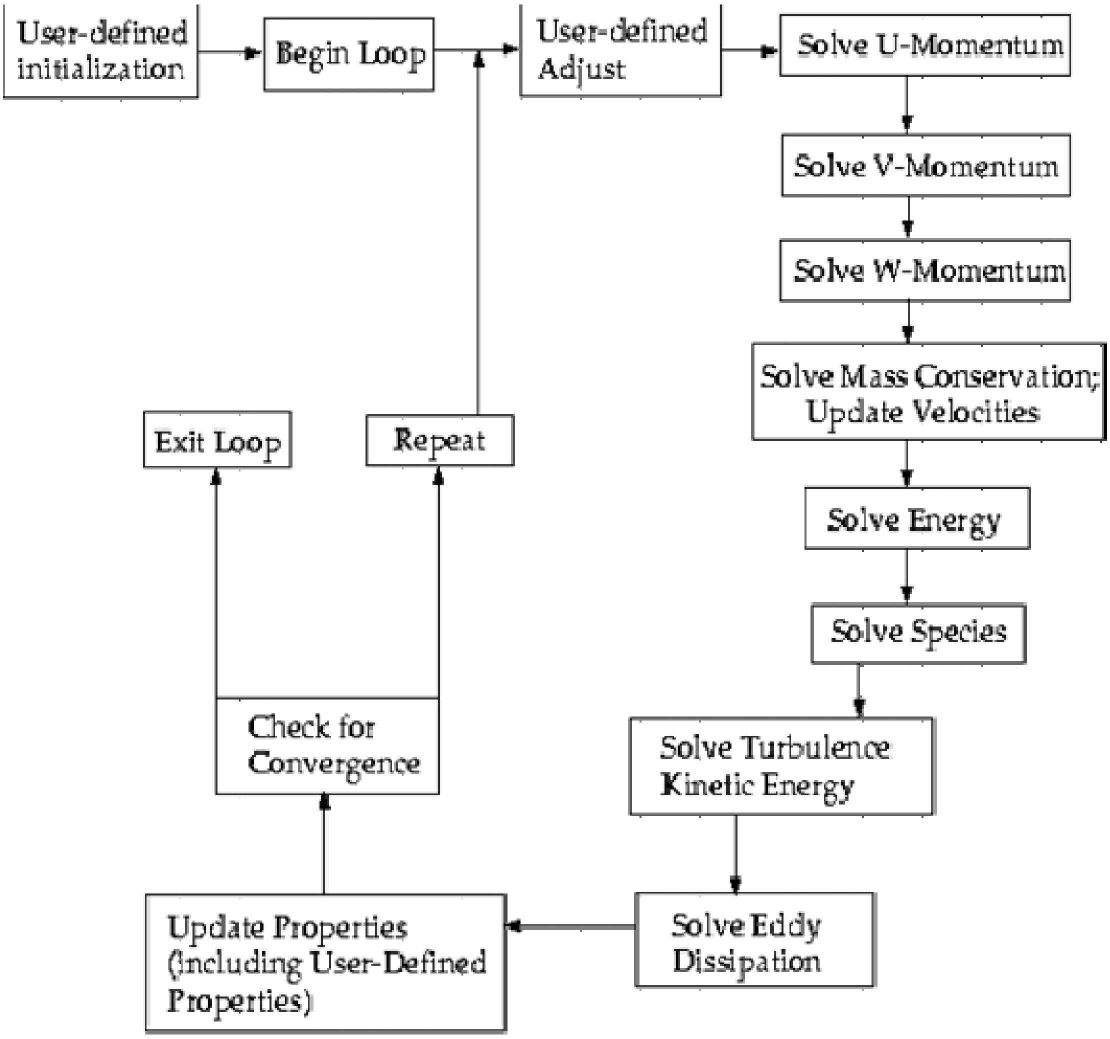


Figure 3.3.1: 分离解算器的解程序

耦合求解器

在耦合求解器求解过程中（Figure 3.3.2）,用户定义的初始化函数（使用 DEFINE_INIT 定义的）在迭代循环开始之前执行。然后，迭代循环开始执行用户定义的调整函数（使用 DEFINE_ADJUST 定义的）。接着，FLUENT 求解连续、动量和（适合的地方）能量的控制方程和同时地一套物质输运或矢量方程。其余的求解步骤与分离式求解器相同（Figure 3.3.1）。

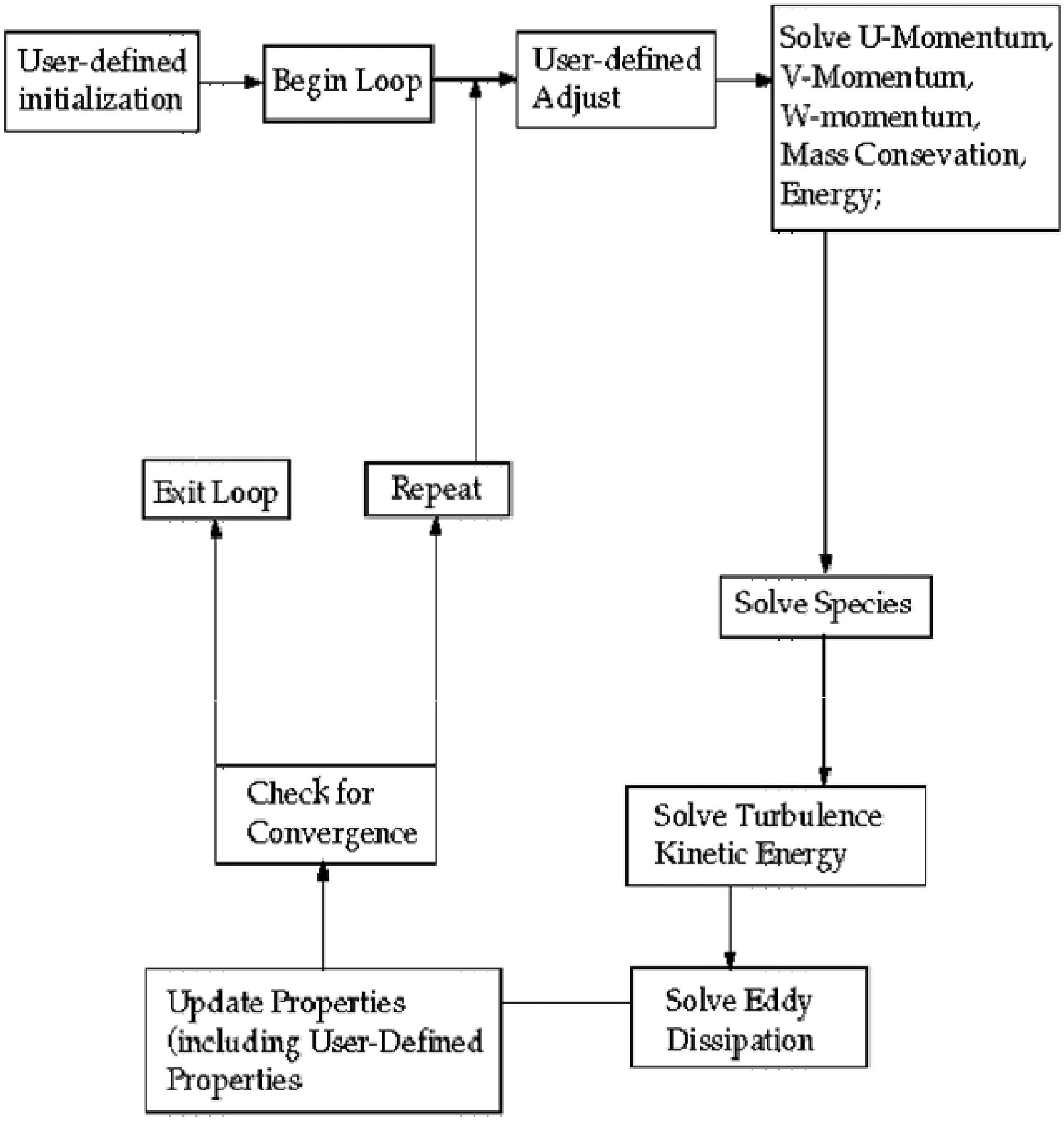


Figure 3.3.2: Solution Procedure for the Coupled Solver

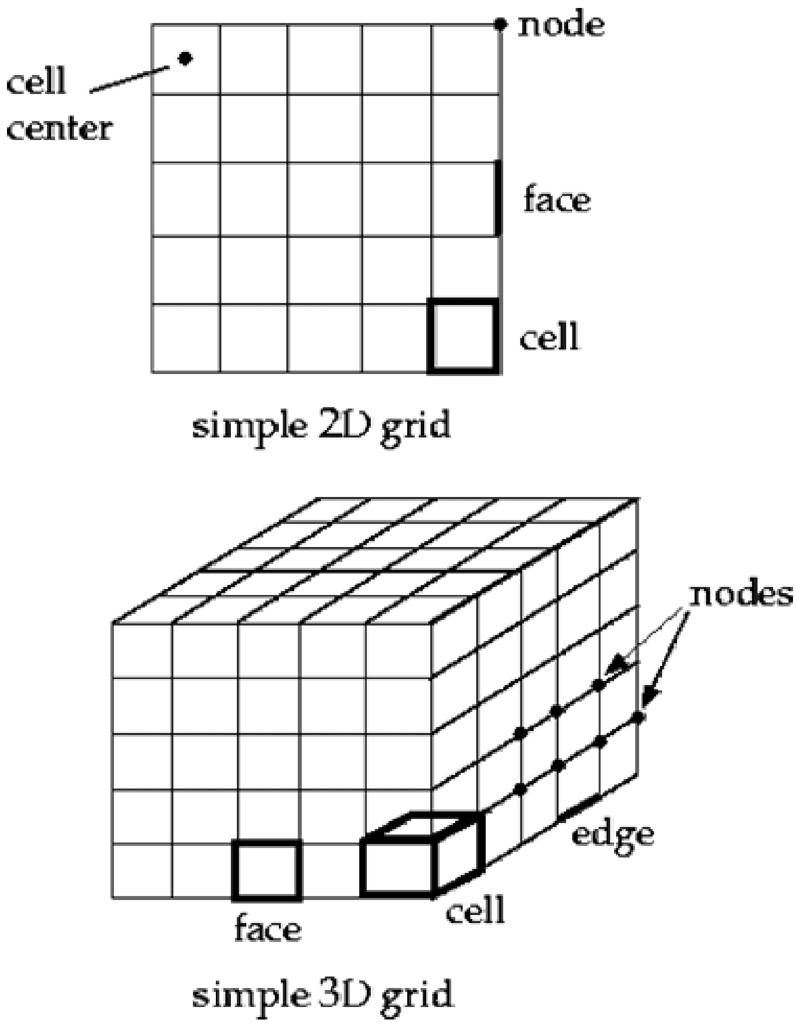
3.4 FLUENT 网格拓扑

在我们开始讨论 FLUENT 特殊的数据类型之前，你必须理解网格拓扑学的术语因为 FLUENT 数据类型是为这些实体定义的。下面是显示在 Figure 3.4.1 中的网格实体的定义。

单元（cell）	区域被分割成的控制容积
单元中心（cell center）	FLUENT 中场数据存储的地方
面（face）	单元（2D or 3D）的边界
边（edge）	面（3D）的边界

节点（node）	网格点
单元线索（cell thread）	在其中分配了材料数据和源项的单元组
面线索（face thread）	在其中分配了边界数据的面组
节点线索（node thread）	节点组
区域（domain）	由网格定义的所有节点、面和单元线索的组合

Figure 3.4.1: Grid Terminology



3.5 FLUENT 数据类型

除了标准的 C 语言数据类型如 real, int 等可用于在你的 UDF 中定义数据外，还有几个 FLUENT 指定的与求解器数据相关的数据类型。这些数据类型描述了 FLUENT 中定义的网格的计算单位（见 Figure 3.4.1）。使用这些数据类型定义的变量既有代表性地补充了 DEFINE macros 的自变量，也补充了其它专门的访问 FLUENT 求解器数据的函数。

一些更为经常使用的 FLUENT 数据类型如下：

cell_t

face_t

Thread

Domain

Node

cell_t 是线索（thread）内单元标识符的数据类型。它是一个识别给定线索内单元的整数下标。face_t 是线索内面标识符的数据类型。它是一个识别给定线索内面的整数下标。

Thread 数据类型是 FLUENT 中的数据结构。它充当了一个与它描述的单元或面的组合相关的数据容器。

Node 数据类型也是 FLUENT 中的数据结构。它充当了一个与单元或面的拐角相关的数据容器。

Domain 数据类型代表了 FLUENT 中最高水平的数据结构。它充当了一个与网格中所有节点、面和单元线索组合相关的数据容器。

!! 注意，FLUENT 中所有数据类型都是 情形敏感的（case-sensitive）。

3.6 使用 DEFINE Macros 定义你的 UDF

Fluent.Inc 为你提供了一套你必须使用它来定义你的 UDF 的预定义函数。这些定义 UDF 的函数在代码中作为宏执行，可在作为 DEFINE(全部大写)宏的文献中查阅。对每个 DEFINE 宏的完整描述和它的应用例子，可参考第四章。

DEFINE 宏的通用格式为：

DEFINE_MACRONAME(udf_name, passed-in variables)

这里括号内第一个自变量是你的 UDF 的名称。名称自变量是情形敏感的必须用小写字母指定。一旦函数被编译（和连接），你为你的 UDF 选择的名字在 FLUENT 下拉列表中将变成可见的和可选的。第二套输入到 DEFINE 宏的自变量是从 FLUENT 求解器传递到你的函数的变量。

在下面的例子中，宏

DEFINE_PROFILE(inlet_x_velocity, thread, index)

用两个从 FLUENT 传递到函数的变量 thread 和 index 定义了名字为 inlet_x_velocity 的分布函数。这些 passed-in 变量是边界条件区域的 ID（作为指向 thread 的指针）而 index 确定

了被存储的变量。一旦 UDF 被编译，它的名字（例如，inlet_x_velocity）将在 FLUENT 适当的边界条件面板（例如，**Velocity Inlet** 面板）的下拉列表中变为可见的和可选的。

!! 注意，所有用于 DEFINE 宏的自变量必须放在你的源代码的同一行上。分割 DEFINE 的声明为几行可能导致编译错误。

3.7 在你的 UDF 源文件中包含 udf.h 文件(Including the udf.h File in Your UDF Source File)

DEFINE 宏的定义位于称为 udf.h(见附录 A 的列表)的头文件中。为了使 DEFINE 宏延伸到编译过程，你必须在你写的每个 UDF 源文件的开始包含 udf.h 文件。

```
#include "udf.h"


/* Always include udf.h when writing a UDF. It translates the DEFINE */
/* and other macros into C, which is what the compiler understands.   */
```

通过在你的 UDF 源文件中包含 udf.h，编译过程中所有的 DEFINE 宏的定义与源代码一起被包含进来。udf.h 文件也为所有的 C 库函数头文件包含#include 指示，与大部分头文件是针对 Fluent 提供的宏和函数是一样的（例如，mem.h）。除非有另外的指示，没必要在你的 UDF 中个别地包含这些头文件。

还有，当你编译你的 UDF 时，你不必放置 udf.h 的拷贝在你的当地目录下；一旦你的 UDF 被编译，FLUENT 求解器会自动地从 Fluent.Inc/fluent6.x/src/目录来读取 udf.h 文件。

举例

从前面部分的宏

DEFINE_PROFILE(inlet_x_velocity, thread, index)

定义在 udf.h 文件中为

#define DEFINE_PROFILE(name, t, i) void name(Thread *t, int i)

在编译过程中延伸为

void inlet_x_velocity(Thread *thread, int index)

名字为 inlet_x_velocity 的函数不返回值由于它被声明为空的数据类型。

3.8 在你的函数中定义变量（Defining Variable in Your Function）

在你的 UDF 源文件中包含了 udf.h 头文件后，你必须定义真实的变量。使用把它们定义在所有函数之外的全局变量（如果它们被源文件中大部分或所有函数共享）是非常方便的。

关于全局变量的信息见 Section 2.5.3。局部于函数的任何变量必须在函数内声明。局部变量的信息见 Section 2.5.2。

3.9 函数体（Functin Body）

你的 UDF 源文件中的 C 函数体被包含在 DEFINE 声明之下的一对大括号内，显示在下面的例子中。在这个例子中，mu_lan 和 temp 是局部变量。只有 cell_viscosity 函数认识它们。

例子

```
DEFINE_PROPERTY(cell_viscosity, cell, thread)
{
  real mu_lam;
  real temp = C_T(cell, thread);

  if (temp > 288.)
    mu_lam = 5.5e-3;
  else if (temp > 286.)
    mu_lam = 143.2135 - 0.49725 * temp;
  else
    mu_lam = 1.;

  return mu_lam;
}
```

3.10　UDF 任务（UDF Tasks）

UDF 可执行的任务有五种不同的类型：

1. 返回值
2. 修改自变量
3. 返回值和修改自变量
4. 修改 FLUENT 变量（不能作为自变量传递）
5. 写信息到（或读取信息从）case 或 data 文件

函数能返回值，除非它们在 udf.h 文件中被定义为 void。如果它们不返回值，它们能修改自变量，修改存储在内存中的变量，或与 case 和 data 文件一起执行输入输出（I/O）任务。

在 Section 3.10.1-3.10.5 中，提供了描述上面提到的五种不同的函数任务中每一种的 UDF 源代码例子。

3.10.1 返回值的函数（Function that Return a Value）

下面的 UDF 是一个返回值到 FLUENT 求解器的函数例子。名为 cell_viscosity 的函数计算了依赖温度的粘度值(mu_lam)并返回这个值到求解器。

```
/*******************************************************************/
/* UDF that returns a value to the solver                          */
/* Specifies a temperature-dependent viscosity property            */
/*******************************************************************/#include
"udf.h"

DEFINE_PROPERTY(cell_viscosity, cell, thread)
{
   real mu_lam;
   real temp = C_T(cell, thread);

   if (temp > 288.)
      mu_lam = 5.5e-3;
   else if (temp > 286.)
      mu_lam = 143.2135 - 0.49725 * temp;
   else
      mu_lam = 1.;

   return mu_lam;
}
```

cell_viscosity 使用了 DEFINE_PROPERTY 宏（在 Section 4.3.6 中描述）来定义。DEFINE_PROPERTY 返回一个 udf.h 中指定的 real 数据类型。两个 real 变量传入函数：通过函数计算的层流粘度 mu_lam；和 C_T(cell,thread)的值，它是在考虑中的单元的温度值。温度值在它下降范围的基础上被检测，mu_lam 的适当值被计算。在函数结尾，mu_lam 的计

算值被返回。

3.10.2 修改自变量的函数（Function that Modify an Argument）

下面的 UDF 是一个修改一个自变量的函数的例子。名字为 user_rate 的函数为一个两种气态物质的的简单系统产生一个自定义的体积反应速率。Real 指针 rr 作为自变量传递给函数，指针指向的变量在函数内被修改。

```
/**********************************************************/
/* UDF that modifies one of its arguments                */
/* Specifies a reaction rate in a porous medium          */
/**********************************************************/

#include "udf.h"

#define K1 2.0e-2
#define K2 5.

DEFINE_VR_RATE(user_rate, c, t, r, mole_weight, species_mf, rr, rr_t)
{
  real s1 = species_mf[0];
  real mw1 = mole_weight[0];

  if (FLUID_THREAD_P(t) && THREAD_VAR(t).fluid.porous)
    *rr = K1*s1/pow((1.+K2*s1),2.0)/mw1;
  else
    *rr = 0.;
}
```

user_rate 使用了 DEFINE_VR _RATE 宏（见 Section 4.3.14）来定义。该函数执行一个当前考虑的单元是否在多孔区域的测试，这个反应速率只应用于多孔区域。real 指针变量 rr 是一个传递给函数的自变量。UDF 使用废弃操作符 * 分配反应速率值给废弃指针 *rr。指针 rr 指向的目标是设置反应速率。通过这个操作，存储在内存中这个指针上的字符的地址被改变

了，不再是指针地址本身。（关于废弃指针的详细内容见[3]）。

3.10.3 返回一个值和修改一个自变量的函数（Functions that Return a Value ans Modify an Argument）

下面的 UDF 是一个修改它的自变量并返回一个值到 FLUENT 求解器的函数例子。名字为 user_swirl 的函数修改 ds 自变量，指定旋转速度源项并返回它到求解器。

```
/*************************************************************/
/* UDF that returns a value and modifies an argument        */
/* Specifies a swirl-velocity source term                    */
/*************************************************************/

#include "udf.h"

#define OMEGA 50.       /* rotational speed of swirler */
#define WEIGHT 1.e20 /* weighting coefficients in linearized equation */

DEFINE_SOURCE(user_swirl, cell, thread, dS, eqn)
{
  real w_vel, x[ND_ND], y, source;

  C_CENTROID(x, cell, thread);
  y = x[1];

  w_vel = y*OMEGA;         /* linear w-velocity at the cell */

  source = WEIGHT*(w_vel - C_WSWIRL(cell,thread));
  dS[eqn] = -WEIGHT;

  return source;
}
```

user_swirl 使用 DEFINE_SOURCE 宏来定义（在 Section 4.3.8 中描述）。DEFINE_SOURCE 返回一个在 udf.h 中指定的数据类型。函数采用自变量 ds（它是数组的名字）并设置由 eqn 指定的元素为关于 *w* 速度（w_vel）导数的值。（这是 z 动量方程源项）。这个函数也计算了旋转速度源项的值 source，并返回这个值到求解器。

3.10.4 修改 FLUENT 变量的函数

下面的 UDF 是一个修改存储在内存中 FLUENT 变量的函数例子。名字为 inlet_x_velocity 的函数使用 F_PROFILE(a Fluent Inc. provided **utility（应用程序）"Translate by 金山词霸")** 来修改存储在内存中的 x 速度分布边界条件。

```
/**********************************************************************/
/* UDF that modifies a FLUENT solver variable                         */
/* Specifies a steady-state velocity profile boundary condition       */
/**********************************************************************/

#include "udf.h"

DEFINE_PROFILE(inlet_x_velocity, thread, index)
{
  real x[ND_ND];        /* this will hold the position vector */
  real y;
  face_t f;

  begin_f_loop(f, thread)
    {
      F_CENTROID(x,f,thread);
      y = x[1];
      F_PROFILE(f, thread, index) = 20. - y*y/(.0745*.0745)*20.;
    }
  end_f_loop(f, thread)
}
```

inlet_x_velocity 使用 DEFINE_PROFLIE 宏来定义（在 Section 4.3.5 中描述）。它的自变量是 thread 和 index。Thread 是一个指向面线索的指针，而 index 是一个每个循环内为变量设置数值标签的整数。DEFINE_PROPERTY 在 udf.h 文件中一个返回 void 的数据类型。

函数由声明变量 f 作为 face_t 数据类型开始。一维数组 x 和变量 y 是 real 数据类型。循环宏用来在区域中每个面上循环以创建型线或数据数组。在每个循环内，F_CENTROID 为含有 index f 的面输出面质心的值（数组 x），index f 在由 thread 指向的线索上。存储在 x[1] 中的 y 坐标分配给变量 y，它用于计算 x 速度。然后这个值分配给 F_PROFILE, 它使用整数 index（由求解器传递个它）来设置内存中面上的 x 速度值。

3.10.5 写入 Case 或 Data 文件或从中读取的函数(Functions that Write to or Read from a Case or Data File)

下面的 C 源代码包含了写信息到 data 文件和读回它的函数例子。这是一个包含多个连接在一起的 UDF 的单个 C 文件例子。

```
/*******************************************************************/
/* UDF that increment a variable, write it to a data file          */
/* and read it back in                                             */
/*******************************************************************/

#include "udf.h"

int kount = 0;    /* define global variable kount */

DEFINE_ADJUST(demo_calc, domain)
{
   kount++;
   printf("kount = %d\n",kount);
}


DEFINE_RW_FILE(writer, fp)
{
```

```
    printf("Writing UDF data to data file...\n");
    fprintf(fp, "%d",kount); /* write out kount to data file */
}


DEFINE_RW_FILE(reader, fp)
{
    printf("Reading UDF data from data file...\n");
    fscanf(fp, "%d",&kount); /* read kount from data file */
}
```

在顶部的列表中，整数 kount 被定义为全局的（由于它被源代码文件中的所有三个函数使用）并初始化为 0。名字为 demo_calc 的第一个函数，使用 DEFINE_SDJUST 宏来定义。（关于 DEFINE_ADJUST 的详细信息见 Section 4.2.1）。在 demo_calc 中，kount 的值每次迭代后增加因为每次迭代调用 DEFINE_ADJUST 一次。名字为 writer 的第二个函数，使用 DEFINE_RW_FILE 宏来定义。（关于 DEFINE_RW_FILE 的详细信息见 Section 4.2.4）。当保存数据文件时，它指示 FLUENT 写当前 kount 值到数据文件。名字为 reader 的第三个函数，当读取数据文件时，它指示 FLUENT 从这个数据文件中读取 kount 的值。

这三个函数一起工作如下。如果你运行 10 次迭代计算（kount 将增加到值为 10）并保存这个数据文件，当前 kount(10)的值被写入你的数据文件。如果你读这个数据返回到 FLUENT 并继续计算，kount 将以值 10 开始随着每次迭代继续增加。注意，你可尽你所想的保存静态变量，但是必须保证以与它们被写的相同顺序来读取它们。

3.11 为多相流应用写 UDF（Writing UDF for Mulutiphase Applications）

当一个多相流模型在 FLUENT 中被激活时，属性和变量的存储和对单相一样应当为所有相的混合设置它们。这可以通过使用附加线索（thread）和区域数据结构在代码中得以表明。

3.11.1 多相应用的数据结构（Data Structure for Multiphase Applications）

区域和线程（Domains and Threads）

在多相流应用中，最高级别的区域被用作超级区域。每相占据的区域用作子区域第三种

类型区域是相互作用区，为了定义相间的相互作用才引入它的。当混合属性和变量必要时（所有相的和），超级区域用于这些数量而子区域携带了单相的信息。在单一相中混合的概念用于描述所有种类（成分）的和而在多相中它描述了所有相的总和。这个区别是非常重要的因为以后代码将延伸到多相多组分（例如，这里相是种类的混合）。

由于求解器的数据存储在线程（thread）数据结构中，线程必须既和超级区域相联系又和每个子区域相联系。就是说，对定义在超级区域上的每个单元或面线程，有相应的单元或面线程定义在每个子区域上。定义在超级区域每个线程上的某些信息与子区域每个相应线程上的共享。与超级区域相关的线程被用作超级线程，而与子区域相关的线程被用作相级别线程或子线程。区域和线程的层次总结在 Figure 3.11.1 中。

Figure 3.11.1: Domain and Thread Structure Hierarchy

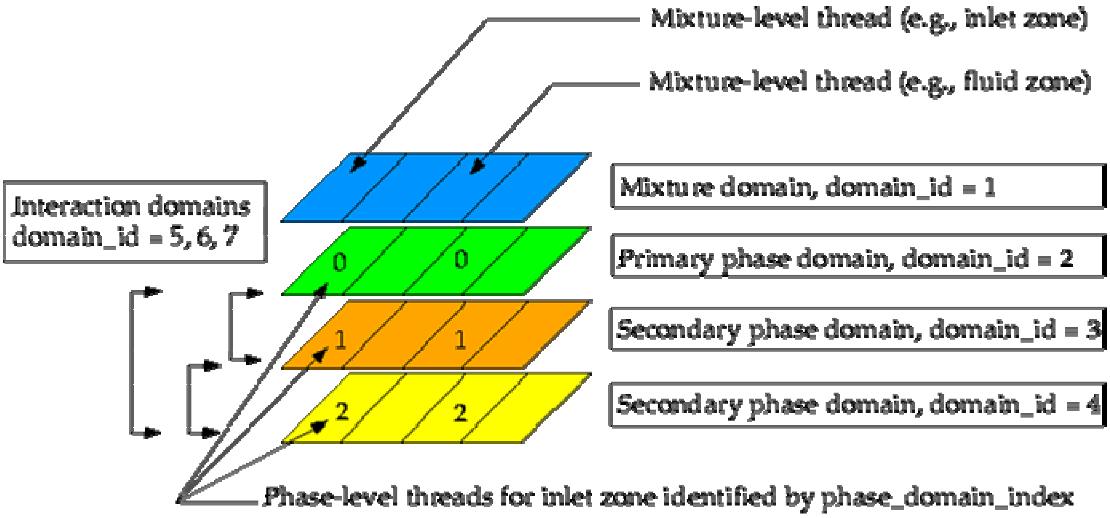


Figure 3.11.1 也引入了 domain_id 和 phase_domain_indexed 的概念。domain_id 在 UDF 中用于辨别超级区域从主要和次要的相级别区域中。超级区域的 domain_id 值总是被指定为 1。相互作用区域也有相同的 domain_id。domain_ids 不必要如 Figure3.11.1 显示的顺序排列。dhase_domain_index 在 UDF 中用于从次要相级别线程中辨别主要相级别线程。对主要相级别线程，phase_domain_index 总是分配值为 0。

访问数据（Accessing Data）

当你写 UDF 或为多相应用使用宏时，参考与尝试访问其属性的相（subdomain）或混合（super domain）相关的数据结构（thread or domain）是很重要的。作为例子，宏 C_R(c,t) 将返回线程 t 的单元 c 上的密度。如果 t 是指向超级线程的指针，那么返回的是混合密度。如果 t 是指向子线程的指针，那么返回的是相密度。

当传递到你的 UDF 的线程或区域指针不被你的函数需要时，也有一些例子。这取决于你使用的多相模型，你尝试修改的属性或项（例如，你使用的那个 DEFINE 宏），还有受到影响的相或混合。为了更好地理解这点，回想一个混合模型和欧拉多相模型之间区别的例子。在混合模型中，为混合相求解单一的动量方程，混合相的属性由它的相的总和来决定。在欧拉模型中，动量方程为每一相求解。当使用混合模型时，FLUENT 允许你直接为混合相指定动量源项（使用 DEFINE_SOURCE），但是不能为欧拉模型。对后者，你可以为单个相指定动量源项。因此，多相模型及被 UDF 修改的项，决定了那哪个区域或线程是需要的。从求解器传递到你的 UDF 的特定的区域或线程的结构取决于你使用的 DEFINE 宏。例如，DEFINE_INIT 和 DEFINE_ADJUST 函数总是传递与超级区域相关的区域结构。DEFINE_ON_DEMAND 函数不能传递任何区域结构。如果你的 UDF 不能明确地传递指针到你的函数定义要求的线程或区域，那么你可以使用 multiphase-specific utility macro 找会它（见第 6 章）。为了你方便使用，表 3.11.1-3.11.6 总结了每个多相模型和相，在该相上为每个给定变量指定了 UDF。从这些信息，你可推断出那些区域结构是从求解器传递到 UDF 的。

单相和多相模型应用 UDF 之间的区别（Differences Between UDF for Single-Phase and Multiphase Applications）

注：在许多例子中，为单相流动写的 UDF 源代码和为多相流动写的是相同的。例如，假设函数只从它被连接（hooked）到的相级别区域访问数据，为单相边界型线（使用 DEFINE_PROFIEL 定义的）写的 C 代码为多相边界型线写的代码之间是没有区别的。然而，如果那些函数从除混合级别区域之外的任何区域访问数据，调整和初始化 UDF 的代码对单相和多相流动是不同的。

3.11.2 对多相模型使用 UDF（Using UDF for Multiphase Models）

在多相模型中，从求解器传递到你的 UDF 的数据结构（例如区域和线程指针）取决于你使用的 DEFINE 宏。传递哪一个特定的区域或线程取决于函数连接到求解器的什么地方。例如，被连接到混合模型的函数传递超级区域结构，而连接到特定相的函数传递子区域结构。表 3.11.1-3.11.6 列举了 DEFINE 和对每个多相模型 UDF 被连接到的相。从这些信息你可推断出那些数据结构被传递。回想 DEFINE_ADJUST 和 DEFINE_INIT UDF are hardwired to the mixture-level domain，而 DEFINE_ON_DEMAND 函数不能连接到任何区域。

Table 3.11.1: `DEFINE` Macro Usage for the VOF Model

Variable	Macro	Phase Specified On
volume fraction	DEFINE_PROFILE	secondary phase(s)
velocity at a boundary	DEFINE_PROFILE	mixture
pressure at a boundary	DEFINE_PROFILE	mixture
boundary temperature	DEFINE_PROFILE	mixture
turbulent kinetic energy	DEFINE_PROFILE	mixture
turbulent dissipation rate	DEFINE_PROFILE	mixture
mass source	DEFINE_SOURCE	primary and
		secondary phase(s)
momentum source	DEFINE_SOURCE	mixture
energy source	DEFINE_SOURCE	mixture
turbulent kinetic energy source	DEFINE_SOURCE	mixture
turbulent dissipation rate source	DEFINE_SOURCE	mixture
density	DEFINE_PROPERTY	primary and
		secondary phase(s)
viscosity	DEFINE_PROPERTY	primary and
		secondary phase(s)

Table 3.11.2: DEFINE Macro Usage for the Mixture Model

Variable	Macro	Phase Specified On
volume fraction	DEFINE_PROFILE	secondary phase(s)
velocity at a boundary	DEFINE_PROFILE	primary and
		secondary phase(s)

pressure at a boundary	DEFINE_PROFILE	mixture
boundary temperature	DEFINE_PROFILE	mixture
turbulent kinetic energy	DEFINE_PROFILE	mixture
turbulent dissipation rate	DEFINE_PROFILE	mixture
mass source	DEFINE_SOURCE	primary and
		secondary phase(s)
momentum source	DEFINE_SOURCE	mixture
energy source	DEFINE_SOURCE	mixture
turbulent kinetic energy source	DEFINE_SOURCE	mixture
turbulent dissipation rate source	DEFINE_SOURCE	mixture
density	DEFINE_PROPERTY	primary and
		secondary phase(s)
viscosity	DEFINE_PROPERTY	primary and
		secondary phase(s)
diameter	DEFINE_PROPERTY	secondary phase(s)
slip velocity	DEFINE_VECTOR_	phase interaction
	EXCHANGE_PROPERTY	

Table 3.11.3: DEFINE Macro Usage for the Eulerian Model (Laminar Flow)

Variable	**Macro**	**Phase Specified On**

volume fraction	DEFINE_PROFILE	secondary phase(s)
velocity at a boundary	DEFINE_PROFILE	primary and
		secondary phase(s)
pressure at a boundary	DEFINE_PROFILE	mixture
granular temperature	DEFINE_PROFILE	secondary phase(s)
mass source	DEFINE_SOURCE	primary and
		secondary phase(s)
momentum source	DEFINE_SOURCE	primary and
		secondary phase(s)
density	DEFINE_PROPERTY	primary and
		secondary phase(s)
viscosity	DEFINE_PROPERTY	primary and
		secondary phase(s)
granular diameter	DEFINE_PROPERTY	secondary phase(s)
granular viscosity	DEFINE_PROPERTY	secondary phase(s)
granular bulk viscosity	DEFINE_PROPERTY	secondary phase(s)
granular frictional viscosity	DEFINE_PROPERTY	secondary phase(s)
drag coefficient	DEFINE_EXCHANGE_PROPERTY	phase interaction
lift coefficient	DEFINE_EXCHANGE_PROPERTY	phase interaction

Table 3.11.4: DEFINE Macro Usage for the Eulerian Model (Mixture Turbulence Model)

Variable	**Macro**	**Phase Specified On**
volume fraction	DEFINE_PROFILE	secondary phase(s)
velocity at a boundary	DEFINE_PROFILE	primary and
		secondary phase(s)
pressure at a boundary	DEFINE_PROFILE	mixture
granular temperature	DEFINE_PROFILE	secondary phase(s)
turbulent kinetic energy	DEFINE_PROFILE	mixture
turbulent dissipation rate	DEFINE_PROFILE	mixture
mass source	DEFINE_SOURCE	primary and
		secondary phase(s)
momentum source	DEFINE_SOURCE	primary and
		secondary phase(s)
turbulent kinetic energy source	DEFINE_SOURCE	mixture
turbulent dissip. rate source	DEFINE_SOURCE	mixture
density	DEFINE_PROPERTY	primary and
		secondary phase(s)
viscosity	DEFINE_PROPERTY	primary and
		secondary phase(s)
granular diameter	DEFINE_PROPERTY	secondary phase(s)
granular viscosity	DEFINE_PROPERTY	secondary phase(s)
granular bulk viscosity	DEFINE_PROPERTY	secondary phase(s)
granular frictional viscosity	DEFINE_PROPERTY	secondary phase(s)

drag coefficient	DEFINE_EXCHANGE_PROPERTY	phase interaction
lift coefficient	DEFINE_EXCHANGE_PROPERTY	phase interaction

Table 3.11.5: DEFINE Macro Usage for the Eulerian Model (Dispersed Turbulence Model)

Variable	**Macro**	**Phase Specified On**
volume fraction	DEFINE_PROFILE	secondary phase(s)
velocity at a boundary	DEFINE_PROFILE	primary and
		secondary phase(s)
pressure at a boundary	DEFINE_PROFILE	mixture
granular temperature	DEFINE_PROFILE	secondary phase(s)
turbulent kinetic energy	DEFINE_PROFILE	primary phase
turbulent dissipation rate	DEFINE_PROFILE	primary phase
mass source	DEFINE_SOURCE	primary and
		secondary phase(s)
momentum source	DEFINE_SOURCE	primary and
		secondary phase(s)
turbulent kinetic energy source	DEFINE_SOURCE	primary phase
turbulent dissip. rate source	DEFINE_SOURCE	primary phase
density	DEFINE_PROPERTY	primary and
		secondary phase(s)
viscosity	DEFINE_PROPERTY	primary and

		secondary phase(s)
granular diameter	DEFINE_PROPERTY	secondary phase(s)
granular viscosity	DEFINE_PROPERTY	secondary phase(s)
granular bulk viscosity	DEFINE_PROPERTY	secondary phase(s)
granular frictional viscosity	DEFINE_PROPERTY	secondary phase(s)
drag coefficient	DEFINE_EXCHANGE_PROPERTY	phase interaction
lift coefficient	DEFINE_EXCHANGE_PROPERTY	phase interaction

Table 3.11.6: DEFINE Macro Usage for the Eulerian Model (Per-Phase Turbulence Model)

Variable	**Macro**	**Phase Specified On**
volume fraction	DEFINE_PROFILE	secondary phase(s)
velocity at a boundary	DEFINE_PROFILE	primary and
		secondary phase(s)
pressure at a boundary	DEFINE_PROFILE	mixture
granular temperature	DEFINE_PROFILE	secondary phase(s)
turbulent kinetic energy	DEFINE_PROFILE	primary and
		secondary phase(s)
turbulent dissipation rate	DEFINE_PROFILE	primary and
		secondary phase(s)
mass source	DEFINE_SOURCE	primary and
		secondary phase(s)
momentum source	DEFINE_SOURCE	primary and
		secondary phase(s)

turbulent kinetic energy source	DEFINE_SOURCE	primary and secondary phase(s)
turbulent dissip. rate source	DEFINE_SOURCE	primary and
		secondary phase(s)
density	DEFINE_PROPERTY	primary and
		secondary phase(s)
viscosity	DEFINE_PROPERTY	primary and
		secondary phase(s)
granular diameter	DEFINE_PROPERTY	secondary phase(s)
granular viscosity	DEFINE_PROPERTY	secondary phase(s)
granular bulk viscosity	DEFINE_PROPERTY	secondary phase(s)
granular frictional viscosity	DEFINE_PROPERTY	secondary phase(s)
drag coefficient	DEFINE_EXCHANGE_PROPERTY	phase interaction
lift coefficient	DEFINE_EXCHANGE_PROPERTY	phase interaction

3.12 在并行下使用你的 UDF(Using Your UDF in Para

如果你想在 FLUENT 并行版本中使用 UDF，你必须添加一些额外的代码行到你的 UDF。例如，如果你的 UDF 通过在面上循环计算总和，那么每个处理器需要为它拥有的面计算局部和，然后，在循环结尾，全局和将必须执行。如果这个“并行化”的修改没有进入你的 UDF 代码，那么这时总和常被分解成若干量，那些线程上的序号为零的面的计算节点将被 0 除从而导致浮点错误。

为 real 数全局求和的操作是 PRF_GRSUM1。你必须沿着对每个计算节点的总和添加这个操作到你的代码，例子的代码显示如下。

```
/**********************************************************************/
```

```
/*    Sample code demonstrating parallelizing a UDF                              */
/******************************************************************/



/*    compute local sum on each compute-node    */
a = 0;

begin_f_loop()        /*    loop over faces    */
{
    a += ...;            /*    put your local sum function here    */
}
end_f_loop

a = PRF_GRSUM1(a);    /* compute global sum, and assign it to */
                      /* variable named a                       */
```

第四章 DEFINE 宏

本章介绍了 Fluent 公司所提供的预定义宏，我们需要用这些预定义宏来定义 UDF。在这里这些宏就是指 DEFINE 宏。

本章由如下几节组成：

- 4.1 概述
- 4.2 通用解算器 DEFINE 宏
- 4.3 模型指定 DEFINE 宏
- 4.4 多相 DEFINE 宏
- 4.5 离散相模型 DEFINE 宏

4.1 概述

DEFINE 宏一般分为如下四类：

- 通用解算器
- 模型指定
- 多相
- 离散相模型(DPM)

对于本章所列出的每一个 DEFINE 宏，本章都提供了使用该宏的源代码的例子。很多例子广泛的使用了其它章节讨论的宏，如解算器读取（第五章）和 utilities (Chapter 6)。需要注意的是，并不是本章所有的例子都是可以在 FLUENT 中执行的完整的函数。这些例子只是演示一下如何使用宏。

除了离散相模型 DEFINE 宏之外的所有宏的定义都包含在 udf.h 文件中。离散相模型 DEFINE 宏的定义包含在 dpm.h 文件中。为了方便大家，所有的定义都列于附录 A 中。其实 udf.h 头文件已经包含了 dpm.h 文件，所以在你的 UDF 源代码中就不必包含 dpm.h 文件了。

注意：在你的源代码中，DEFINE 宏的所有参变量必须在同一行，如果将 DEFINE 声明分为几行就会导致编译错误。

4.2 通用解算器 DEFINE 宏

本节所介绍的 DEFINE 宏执行了 FLUENT 中模型相关的通用解算器函数。表 4.2.1 提供了 FLUENT 中 DEFINE 宏，以及这些宏定义的功能和激活这些宏的面板的快速参考向导。每一个 DEFINE 宏的定义都在 udf.h 头文件中，具体可以参考附录 A。

- DEFINE_ADJUST (4.2.1 节)
- DEFINE_INIT (4.2.2 节)

- DEFINE_ON_DEMAND (4.2.3 节)
- DEFINE_RW_FILE (4.2.4 节)

表 4.2.1：通用解算器 DEFINE 宏的快速参考向导		
功能	DEFINE 宏	激活该宏的面板
处理变量	DEFINE_ADJUST	**User-Defined Function Hooks**
初始化变量	DEFINE_INIT	**User-Defined Function Hooks**
异步执行	DEFINE_ON_DEMAND	**Execute On Demand**
读写变量到……	DEFINE_RW_FILE	**User-Defined Function Hooks**
Case 和 data 文件		

- 4.2.1 DEFINE_ADJUST
- 4.2.2 DEFINE_INIT
- 4.2.3 DEFINE_ON_DEMAND
- 4.2.4 DEFINE_RW_FILE

4.2.1 DEFINE_ADJUST

功能和使用方法的介绍

DEFINE_ADJUST 是一个用于调节和修改 FLUENT 变量的通用宏。例如，你可以用 DEFINE_ADJUST 来修改流动变量（如：速度，压力）并计算积分。你可以用它来对某一标量在整个流场上积分，然后在该结果的基础上调节边界条件。在每一步迭代中都可以执行用 DEFINE_ADJUST 定义的宏，并在解输运方程之前的每一步迭代中调用它。参考图 3.3.1 和 3.3.2 for 可以大致了解一下当 DEFINE_ADJUST 被调用时 FLUENT 解的过程

宏	DEFINE_ADJUST (name, d)
参变量类型	Domain *d
返回的功能	void

DEFINE_ADJUST 有两个参变量：name 和 d。name 是你所指定的 UDF 的名字。当你的 UDF 编译并连接时，你的 FLUENT 图形用户界面就会显示这个名字，此时你就可以选择它了。d 是 FLUENT 解算器传给你的 UDF 的变量。

D 是一个指向区域的指针，调节函数被应用于这个区域上。区域变量提供了存取网格中所有单元和表面的线程。对于多相流，由解算器传给函数的区域指针是混合层区域指针。DEFINE_ADJUST 函数不返回任何值给解算器。

例子 1

下面的 UDF 名字是 adjust，它使用 DEFINE_ADJUST 对湍流耗散在整个区域上积分。然后这个值会打印在控制台窗口中。每一步迭代都会调用这个 UDF。它可以作为解释程序或者编译后的 UDF 在 FLUENT 中执行。

```
/******************************************************************/
/*  积分湍流耗散并将其打印到控制台窗口的 UDF                          */
/*******************************************************************/

#include "udf.h"

DEFINE_ADJUST(my_adjust, d)
{
   Thread *t;
   /* Integrate dissipation. */
   real sum_diss=0.;
   cell_t c;

   thread_loop_c (t,d)
   {
     begin_c_loop (c,t)
           sum_diss += C_D(c,t)*
           C_VOLUME(c,t);
     end_c_loop (c,t)
   }

   printf("Volume integral of turbulent dissipation: %g\n", sum_diss);
}
```

例子： 2

下面 UDF 的名字是 adjust_fcn，它用 DEFINE_ADJUST 指定了某一自定义标量是另一自定义标量的梯度的函数。该函数在每一次迭代中都会被调用。它可以作为编译后的 UDF 在 FLUENT 中执行。

```
/*******************************************************************/
/* UDF for defining user-defined scalars and their gradients          */
/*******************************************************************/

#include "udf.h"

DEFINE_ADJUST(adjust_fcn, d)
{
   Thread *t;
   cell_t c;
   real K_EL = 1.0;
```

```
   /* Do nothing if gradient isn't allocated yet. */
   if (! Data_Valid_P())
      return;

   thread_loop_c (t, d)
      {
      if (FLUID_THREAD_P(t))
         {
            begin_c_loop_all (c,t)
              {
                C_UDSI(c,t,1)                                     +=
K_EL*NV_MAG2(C_UDSI_G(c,t,0))*C_VOLUME(c,t);
              }
            end_c_loop_all (c, t)
         }
      }
}
```

Activating an Adjust UDF in FLUENT

在为 adjust UDF 的源代码进行编译和连接之后，你可以在 FLUENT 中的 **User-Defined Function Hooks** 面板激活这个函数。更详细的内容请参阅 8.1.1 节。

4.2.2 DEFINE_INIT

功能和使用方法的介绍

你可以用 DEFINE_INIT 宏来定义一组解的初始值。DEFINE_INIT 完成和修补一样的功能，只是它以另一种方式——UDF 来完成。每一次初始化时 DEFINE_INIT 函数都会被执行一次，并在解算器完成默认的初始化之后立即被调用。因为它是在流场初始化之后被调用的，所以它最常用于设定流动变量的初值。参考图 3.3.1 和 3.3.2 关于 FLUENT 解过程的介绍可以看出什么时候调用 DEFINE_INIT 函数。

Macro:	DEFINE_INIT (name, d)
Argument types:	Domain *d
Function returns:	void

DEFINE_INIT 有两个参变量：name 和 d。name 是你所指定的 UDF 的名字。当你的 UDF 编译并连接时，你的 FLUENT 图形用户界面就会显示这个名字，此时你就可以选择它了。d 是 FLUENT 解算器传给你的 UDF 的变量。

d is a pointer to the domain over which the initialization function is to be applied. The domain argument provides access to all cell and face threads in the mesh. For multiphase flows, the domain pointer that is passed to the function by the solver is the mixture-level domain pointer. A DEFINE_INIT function does not return a value to the solver.

例子

下面的 UDF 名字是 my_init_func，它在某一个解中初始化了流动变量。在解过程开始时它被执行了一次。它可以作为解释程序或者编译后的 UDF 在 FLUENT 中执行。

```
/*******************************************************************/
/* UDF for initializing flow field variables                       */
/*********************************************************************
***/

#include "udf.h"

DEFINE_INIT(my_init_function, domain)
{
   cell_t c;
   Thread *t;
   real xc[ND_ND];

   /* loop over all cell threads in the domain    */
   thread_loop_c (t,domain)
      {

         /* loop over all cells    */
         begin_c_loop_all (c,t)
            {
               C_CENTROID(xc,c,t);
               if (sqrt(ND_SUM(pow(xc[0] - 0.5,2.),
                                        pow(xc[1] - 0.5,2.),
                                        pow(xc[2] - 0.5,2.))) < 0.25)
                  C_T(c,t) = 400.;
               else
                  C_T(c,t) = 300.;
            }
         end_c_loop_all (c,t)
      }
}
```

The macro ND_SUM(a, b, c) that is used in the UDF computes the sum of the first two arguments (2D) or all three arguments (3D). It is useful for writing functions involving vector operations so that the same function can be used for 2D and 3D. For a 2D case, the third argument is ignored. See Chapter 5 for a description of predefined solver access macros (e.g., C_CENTROID) and Chapter 6 for utility macros (e.g., ND_SUM).

Activating an Initialization UDF in FLUENT

编译并连接 UDF 源代码之后。you can activate the function in the **User-Defined Function Hooks** panel in **FLUENT**. See Section 8.1.2 for more details.

4.2.3 DEFINE_ON_DEMAND

功能和使用方法的介绍

你可以使用 DEFINE_ON_DEMAND macro to define a UDF to execute on demand in **FLUENT**, rather than having **FLUENT** call it automatically during the calculation. Your UDF will be executed immediately, once it is activated, but it is not accessible while the solver is iterating. Note that the domain pointer d is not explicitly passed as an argument to DEFINE_ON_DEMAND. Therefore, if you want to use the domain variable in your on-demand function, you will need to first retrieve it using the Get_Domain utility provided by Fluent (shown in 例子： below). See Section 6.5.1 for details on Get_Domain.

Macro:	DEFINE_ON_DEMAND (name)
Argument types:	none
Function returns:	void

There is only one argument to DEFINE_ON_DEMAND: name. name is the name of the UDF, specified by you. 当你的 UDF 编译和连接时，你为函数所选择的名字会在 FLUENT 图形用户界面中变得可见，且可被选择。 A DEFINE_ON_DEMAND function does not return a value to the solver.

例子：

下面的 UDF 名字为 demand_calc，计算并打印出当前数据场的最小、最大和平均温度。It then computes a temperature function

$$f(T) = \frac{T - T_{min}}{T_{max} - T_{min}}$$

and stores it in user-defined memory location 0 (which is allocated as described in Section 6.7). Once you execute the UDF (as described in Section 8.1.3), the field values for *f*(*T*) will be available in the drop-down lists in postprocessing panels in **FLUENT**. You can select this field by choosing **udm-0** in the **User Defined Memory...** category. If you write a data file after executing the UDF, the user-defined memory field will be saved to the data file. The UDF can be executed as an interpreted or compiled UDF in **FLUENT**.

```
/*********************************************************************
**/
/* UDF to calculate temperature field function and store in              */
/* user-defined memory. Also print min, max, avg temperatures.           */
/*********************************************************************
**/

#include "udf.h"

DEFINE_ON_DEMAND(on_demand_calc)

Domain *d;     /* declare domain pointer since it is not passed a   */
               /* argument to DEFINE macro                              */
{
   real tavg = 0.;
   real tmax = 0.;
   real tmin = 0.;
   real temp,volume,vol_tot;
   Thread *t;
   cell_t c;
   d = Get_Domain(1);        /* Get the domain using Fluent utility */

   /* Loop over all cell threads in the domain */
   thread_loop_c(t,d)
     {

     /* Compute max, min, volume-averaged temperature */

     /* Loop over all cells   */
     begin_c_loop(c,t)
       {
         volume = C_VOLUME(c,t);     /* get cell volume */
         temp = C_T(c,t);                /* get cell temperature */
```

```
            if (temp < tmin || tmin == 0.) tmin = temp;
            if (temp > tmax || tmax == 0.) tmax = temp;

            vol_tot += volume;
            tavg += temp*volume;

          }
        end_c_loop(c,t)

        tavg /= vol_tot;

        printf("\n Tmin = %g      Tmax = %g      Tavg = %g\n",tmin,tmax,tavg);

        /* Compute temperature function and store in user-defined memory*/
        /*(location index 0)                                              */

        begin_c_loop(c,t)
          {
            temp = C_T(c,t);
            C_UDMI(c,t,0) = (temp-tmin)/(tmax-tmin);
          }
        end_c_loop(c,t)

        }
}
```

Get_Domain is a macro that retrieves the pointer to a domain. It is necessary to get the domain pointer using this macro since it is not explicitly passed as an argument to DEFINE_ON_DEMAND. The function, named on_demand_calc, does not take any explicit arguments. Within the function body, the variables that are to be used by the function are defined and initialized first. Following the variable declarations, a looping macro is used to loop over each cell thread in the domain. Within that loop another loop is used to loop over all the cells. Within the inner loop, the total volume and the minimum, maximum, and volume-averaged temperature are computed. These computed values are printed to the **FLUENT** console. Then a second loop over each cell is used to compute the function *f*(*T*) and store it in user-defined memory location 0. Refer to Chapter 5 for a description of predefined solver access macros (e.g., C_T) and Chapter 6 for utility macros (e.g., begin_c_loop).

Activating an On-Demand UDF in FLUENT

After you have compiled and linked the source code for your on-demand UDF, you can activate the function in the **Execute On Demand** panel in **FLUENT**. See Section 8.1.3 for more details.

4.2.4 DEFINE_RW_FILE

功能和使用方法的介绍

你可以使用 DEFINE_RW_FILE macro to define customized information that you want to be written to a case or data file, or read from a case or data file. You can save and restore custom variables of any data types (e.g., integer, real, Boolean, structure) using DEFINE_RW_FILE. It is often useful to save dynamic information (e.g., number of occurrences in conditional sampling) while your solution is being calculated, which is another use of this function. Note that the read order and the write order must be the same when you use this function.

Macro:	DEFINE_RW_FILE (name, fp)
Argument types:	FILE *fp
Function returns:	void

There are two arguments to DEFINE_RW_FILE: name and fp. name is the name of the UDF, specified by you. 当你的 UDF 编译和连接时，你为函数所选择的名字会在 FLUENT 图形用户界面中变得可见，且可被选择。 fp is a variable that is passed by the **FLUENT** solver to your UDF.

fp is a pointer to the file to or from which you are writing or reading. A DEFINE_RW_FILE function does not return a value to the solver.

!! Do not use the fwrite macro in DEFINE_RW_FILE functions that are running on Windows platforms. Use fprintf instead.

例子：

The following C source code contains 例子：s of functions that write information to a data file and read it back. These functions are concatenated into a single file that can be executed as interpreted or compiled in **FLUENT**.

```
/**********************************************************************
***/
/* UDFs that increment a variable, write it to a data file              */
/* and read it back in                                                   */
/**********************************************************************
***/

#include "udf.h"
```

```
int kount = 0;    /* define global variable kount */

DEFINE_ADJUST(demo_calc, domain)
{
   kount++;
   printf("kount = %d\n",kount);
}
DEFINE_RW_FILE(writer, fp)
{
   printf("Writing UDF data to data file...\n");
   fprintf(fp, "%d",kount); /* write out kount to data file */
}
DEFINE_RW_FILE(reader, fp)
{
   printf("Reading UDF data from data file...\n");
   fscanf(fp, "%d",&kount); /* read kount from data file */
}
```

At the top of the listing, the integer kount is defined and initialized to zero. The first function (demo_calc)ltindexdemo_calc is an ADJUST function that increments the value of kount at each iteration, since the ADJUST function is called once per iteration. (See Section 4.2.1 for more information about ADJUST functions.) The second function (writer) instructs **FLUENT** to write the current value of kount to the data file, when the data file is saved. The third function (reader) instructs **FLUENT** to read the value of kount from the data file, when the data file is read.

The functions work together as follows. If you run your calculation for, say, 10 iterations (kount has been incremented to a value of 10) and save the data file, then the current value of kount (10) will be written to your data file. If you read the data back into **FLUENT** and continue the calculation, kount will start at a value of 10 and will be incremented at each iteration. Note that you can save as many static variables as you want, but you must be sure to read them in the same order in which they are written.

Activating a Read/Write Case or Data File UDF in FLUENT

After you have compiled and linked the source code for your read/write UDF, you can activate the function in the **User-Defined Function Hooks** panel in **FLUENT**. See Section 8.1.4 for more details.

4.3 模型指定 DEFINE 宏

本节所介绍的 DEFINE 宏用于 set parameters for a particular model in **FLUENT**. Table 4.3.1 provides a quick reference guide to the DEFINE macros, the functions

they are used to define, and the panel where they are activated in **FLUENT**. Definitions of each DEFINE macro are listed in the udf.h header file. For your convenience, the definitions are also provided in Appendix A.

- DEFINE_DELTAT (Section 4.3.1)
- DEFINE_DIFFUSIVITY (Section 4.3.2)
- DEFINE_HEAT_FLUX (Section 4.3.3)
- DEFINE_NOX_RATE (Section 4.3.4)
- DEFINE_PROFILE (Section 4.3.5)
- DEFINE_PROPERTY(Section 4.3.6)
- DEFINE_SCAT_PHASE_FUNC (Section 4.3.7)
- DEFINE_SOURCE (Section 4.3.8)
- DEFINE_SR_RATE (Section 4.3.9)
- DEFINE_TURB_PREMIX_SOURCE (Section 4.3.10)
- DEFINE_TURBULENT_VISCOSITY (Section 4.3.11)
- DEFINE_UDS_FLUX (Section 4.3.12)
- DEFINE_UDS_UNSTEADY (Section 4.3.13)
- DEFINE_VR_RATE (Section 4.3.14)

Table 4.3.1: Quick Reference Guide for Model-Specific DEFINE Macros

Variable	DEFINE **Macro**	**Panel Activated In**
species mass fraction	DEFINE_PROFILE	boundary condition
		(e.g., **Velocity Inlet**)
velocity at a boundary	DEFINE_PROFILE	boundary condition
pressure at a boundary	DEFINE_PROFILE	boundary condition
boundary temperature	DEFINE_PROFILE	boundary condition
turbulent kinetic energy	DEFINE_PROFILE	boundary condition
turbulent dissipation rate	DEFINE_PROFILE	boundary condition
mass source	DEFINE_SOURCE	boundary condition
momentum source	DEFINE_SOURCE	boundary condition
energy source	DEFINE_SOURCE	boundary condition
turbulent k.e. source	DEFINE_SOURCE	boundary condition
turb. dissipation rate source	DEFINE_SOURCE	boundary condition

UDS or species mass diffusivity	DEFINE_DIFFUSIVITY	**Materials**
turbulent viscosity	DEFINE_TURBULENT_VISCOSITY	**Viscous Model**
heat flux	DEFINE_HEAT_FLUX	boundary condition
density	DEFINE_PROPERTY	**Materials**
viscosity	DEFINE_PROPERTY	**Materials**
mass diffusivity	DEFINE_PROPERTY	**Materials**
thermal conductivity	DEFINE_PROPERTY	**Materials**
absorption coefficient	DEFINE_PROPERTY	**Materials**
scattering coefficient	DEFINE_PROPERTY	**Materials**
laminar flow speed	DEFINE_PROPERTY	**Materials**
rate of strain	DEFINE_PROPERTY	**Materials**
scattering phase function	DEFINE_SCAT_PHASE_FUNC	**Materials**
surface reaction rate	DEFINE_SR_RATE	**User-Defined Function**
		Hooks
volume reaction rate	DEFINE_VR_RATE	**User-Defined Function**
		Hooks
turbulent premixed source	DEFINE_TURB_PREMIX_SOURCE	**User-Defined Function**
		Hooks
scalar flux function	DEFINE_UDS_FLUX	**User-Defined Scalars**
scalar unsteady function	DEFINE_UDS_UNSTEADY	**User-Defined Scalars**
center of gravity motion	DEFINE_CG_MOTION	**Dynamic Zones**
grid motion	DEFINE_GRID_MOTION	**Dynamic Zones**
geometry deformation	DEFINE_GEOM	**Dynamic Zones**
NOx formation rate	DEFINE_NOX_RATE	**NOx Model**
time step (for time	DEFINE_DELTAT	**Iterate**

dependent solutions)		

- 4.3.1 DEFINE_DELTAT
- 4.3.2 DEFINE_DIFFUSIVITY
- 4.3.3 DEFINE_HEAT_FLUX
- 4.3.4 DEFINE_NOX_RATE
- 4.3.5 DEFINE_PROFILE
- 4.3.6 DEFINE_PROPERTY
- 4.3.7 DEFINE_SCAT_PHASE_FUNC
- 4.3.8 DEFINE_SOURCE
- 4.3.9 DEFINE_SR_RATE
- 4.3.10 DEFINE_TURB_PREMIX_SOURCE
- 4.3.11 DEFINE_TURBULENT_VISCOSITY
- 4.3.12 DEFINE_UDS_FLUX
- 4.3.13 DEFINE_UDS_UNSTEADY
- 4.3.14 DEFINE_VR_RATE

4.3.1 DEFINE_DELTAT

功能和使用方法的介绍

你可以使用 DEFINE_DELTAT 宏来控制时间相关问题解的时间步长。This macro can only be used if the adaptive time-stepping method option has been activated in the **Iterate** panel in **FLUENT**.

Macro:	DEFINE_DELTAT (name, domain)
Argument types:	Domain *domain
Function returns:	real

There are two arguments to DEFINE_DELTAT: name and domain. name is the name of the UDF, specified by you. 当你的 UDF 编译和连接时，你为函数所选择的名字会在 FLUENT 图形用户界面中变得可见，且可被选择。 domain is passed by the **FLUENT** solver to your UDF. Your UDF will need to return the real value of the physical time step to the solver.

例子：

下面的 UDF 名字为 mydeltat, is a simple function that shows how you can use DEFINE_DELTAT to change the value of the time step in a simulation. First, RP_Get_Real is used to get the value of the current simulation time (flow_time). Then, for the first 0.5 seconds of the calculation, a time step of 0.1 is set. A time step of 0.2 is set for the remainder of the simulation. The time step variable is then returned to the solver. See Section 6.9 for details on RP_Get_Real.

```
/*******************************************************************
***/
/* UDF that changes the time step value for a time-dependent solution    */
/*******************************************************************
***/
#include "udf.h"

DEFINE_DELTAT(mydeltat, domain)
{
    real time_step;
    real flow_time = RP_Get_Real("flow-time");
    if (flow_time < 0.5)
        time_step = 0.1;
    else
        time_step = 0.2;
    return time_step;
}
```

Activating an Adaptive Time Step UDF in FLUENT

Once you have compiled and linked the source code for an adaptive time step UDF, you can activate it in the **Iterate** panel in **FLUENT**. See Section 8.2.8 for more details.

4.3.2 DEFINE_DIFFUSIVITY

功能和使用方法的介绍

你可以使用 DEFINE_DIFFUSIVITY macro to specify the diffusivity for the species transport equations or user-defined scalar (UDS) transport equations.

Macro:	DEFINE_DIFFUSIVITY (name, c, t, i)
Argument types:	cell_t c

	Thread *t
	int i
Function returns:	real

There are four arguments to DEFINE_DIFFUSIVITY: name, c, and t, and i. name is the name of the UDF, specified by you. 当你的 UDF 编译和连接时，你为函数所选择的名字会在 FLUENT 图形用户界面中变得可见，且可被选择。 c, t, and i are variables that are passed by the **FLUENT** solver to your UDF.

c is an index that identifies a cell within the given thread. t is a pointer to the thread on which the diffusivity function is to be applied. i is an index that identifies the species or user-defined scalar. Your UDF will need to return the real value of diffusivity to the solver.

Note that diffusivity UDFs (defined using DEFINE_DIFFUSIVITY) are called by **FLUENT** from within a loop on cell threads. Consequently, your UDF will not need to loop over cells in a thread since **FLUENT** is doing it outside of the function call. Your UDF will be required to compute the diffusivity only for a single cell and return the real value to the solver.

例子：

下面的 UDF 名字为 mean_age_diff, computes the diffusivity for the mean age of air using a user-defined scalar. Note that the mean age of air calculations do not require that energy, radiation, or species transport calculations have been performed. You will need to set uds-0 = 0.0 at all inlets and outlets in your model. This function can be executed as an interpreted or compiled UDF.

```
/*********************************************************************
**/
/* UDF that computes diffusivity for mean age using a user-defined      */
/* scalar.                                                               */
/*********************************************************************
**/

#include "udf.h"

DEFINE_DIFFUSIVITY(mean_age_diff, c, t, i)
{
	return C_R(c,t) * 2.88e-05 + C_MU_EFF(c,t) / 0.7;
}
```

Activating a Diffusivity UDF in FLUENT

Once you have compiled and linked your diffusivity UDF, you can activate it by selecting it as the mass or UDS diffusivity in the **Materials** panel in **FLUENT**. See Section 8.2.4 for more details.

4.3.3 DEFINE_HEAT_FLUX

功能和使用方法的介绍

In spite of its name, the DEFINE_HEAT_FLUX macro is *not* to be used to explicitly set the heat flux along a wall. **FLUENT** computes the heat flux along a wall based on currently selected models to account for the diffusive and radiative energy fluxes (if any). You must only use a DEFINE_HEAT_FLUX UDF when you want to employ some other heat transfer mechanism that is not currently being modeled. The total heat flux at the wall will be the sum of the currently computed heat flux (based on the activated models) and the heat flux defined by the UDF.

Macro:	DEFINE_HEAT_FLUX (name, f, t, c0, t0, cid, cir)
Argument types:	face_t f
	Thread *t
	cell_t c0
	Thread *t0
	real cid[]
	real cir[]
Function returns:	void

There are seven arguments to DEFINE_HEAT_FLUX: name, f, t, c0, t0, cid, and cir. name is the name of the UDF, specified by you. 当你的 UDF 编译和连接时，你为函数所选择的名字会在 FLUENT 图形用户界面中变得可见，且可被选择。f, t, c0, t0, cir[], and cid[] are variables that are passed by the **FLUENT** solver to your UDF.

f is an index that identifies a wall face within the given thread. t is a pointer to the thread on which the heat flux function is to be applied. c0 is an index that identifies the cell next to the wall, and t0 is a pointer to the adjacent cell's thread.

cid[] and cir[] are real arrays that need to be computed by your UDF. Array cid[] stores the fluid-side diffusive heat transfer coefficients, while array cir[] stores radiative heat transfer coefficients. With these inputs provided to the function, the

diffusive heat flux (qid) and radiative heat flux (qir) are computed by **FLUENT** according to the following equations:

qid = cid[0] + cid[1]*C_T(c0,t0) - cid[2]*F_T(f,t) - cid[3]*pow(F_T(f,t),4)
qir = cir[0] + cir[1]*C_T(c0,t0) - cir[2]*F_T(f,t) - cir[3]*pow(F_T(f,t),4)

The sum of qid and qir defines the total heat flux from the fluid to the wall (this direction being positive flux), and, from an energy balance at the wall, equals the heat flux of the surroundings (exterior to the domain). Note that heat flux UDFs (defined using DEFINE_HEAT_FLUX) are called by **FLUENT** from within a loop over wall faces.

!! In order for the solver to compute C_T and F_T, the values you supply to cid[1] and cid[2] should never be zero.

例子：

Section 10.5.2 provides an 例子： of the P-1 radiation model implementation through a user-defined scalar. An 例子： of the usage of the DEFINE_HEAT_FLUX macro is included in that implementation.

Activating a Heat Flux UDF in FLUENT

Once you have compiled and linked your heat flux UDF, you can activate it by selecting it in the **User-Defined Function Hooks** panel in **FLUENT**. See Section 8.2.2 for more details.

4.3.4 DEFINE_NOX_RATE

功能和使用方法的介绍

你可以使用 DEFINE_NOX_RATE macro to calculate NOx production and reduction rates in **FLUENT**. The UDF rate that you specify is independent of the standard NOx model options. You can deselect the standard NOx options in your simulation, and choose the UDF rate instead.

Macro:	DEFINE_NOX_RATE (name, c, t, NOx)
Argument types:	cell_t c
	Thread *t
	NOx_Parameter *NOx
Function returns:	void

There are four arguments to DEFINE_NOX_RATE: name, c, t, and NOx. name is the name of the UDF, specified by you. 当你的 UDF 编译和连接时，你为函数所选择的名字会在 FLUENT 图形用户界面中变得可见，且可被选择。 c, t, and NOx are variables that are passed by the **FLUENT** solver to your UDF.

c is an index that identifies a cell within the given thread. t is a pointer to the thread on which the NOx rate is to be applied. NOx is a pointer to the NOx structure. A DEFINE_NOX_RATE function does not return a value. The calculated NOx rates are returned through the NOx structure.

Note that, although the data structure is called NOx, the DEFINE_NOX_RATE macro can be used to calculate the rates of any of the pollutant species (i.e., NO, HCN, and NH_3), depending on which of the pollutant species equations is being solved.

例子：

下面编译的 UDF 名字为 user_nox, computes NOx production and reduction rates based on the forward and reverse rates of NO defined as

$$R_f = \frac{2[O]k_1k_2[O_2][N_2]}{k_2[O_2] + k_{-1}[NO]} \qquad (4.3.1)$$

and

$$R_r = \frac{k_{-1}k_{-2}[NO]^2}{k_2[O_2] + k_{-1}[NO]} \qquad (4.3.2)$$

where the rate coefficients, which have units of m^3/mol-s, are defined as

$$k_1 = 1.8 \times 10^8 \exp\left(\frac{-38370}{T}\right) \qquad (4.3.3)$$

$$k_{-1} = 3.8 \times 10^7 \exp\left(\frac{-425}{T}\right) \qquad (4.3.4)$$

$$k_2 = 1.8 \times 10^4 T \exp\left(\frac{-4680}{T}\right) \qquad (4.3.5)$$

$$k_{-2} = 3.8 \times 10^{4} T \exp\left(\frac{-20820}{T}\right) \quad (4.3.6)$$

O concentration is given by

$$[O] = 3.664 \times 10^{1} T^{0.5} [O_2]^{0.5} \exp\left(\frac{-27123}{T}\right) \quad (4.3.7)$$

All concentrations in the rate expressions have units of mol/mol ^{3}.

```
/***********************************************************/
/* UDF  例子：  of User-Defined NOx Rate                    */
/***********************************************************/

#include "udf.h"

#define SMALL_S 1.e-29

DEFINE_NOX_RATE(user_nox, c, t, NOx)
{
    real kf1, kr1, kf2, kr2;
    real o_eq;
    real s1, s2, s3, rf, rr;

    Rate_Const K_F[2] = {{1.80e8, 0.0, 38370.0},
                         {1.80e4, 1.0, 4680.0}};

    Rate_Const K_R[2] = {{3.80e7, 0.0, 425.0},
                         {3.80e3, 1.0, 20820.0}};

    Rate_Const K_O = {3.664e1, 0.5, 27123.0};

    if (NOX_EQN(NOx) != EQ_NO) return;

    kf1 = ARRH(NOx, K_F[0]);
    kr1 = ARRH(NOx, K_R[0]);
    kf2 = ARRH(NOx, K_F[1]);
    kr2 = ARRH(NOx, K_R[1]);
```

```
    s1 = kf2*MOLECON(NOx, O2);
    s3 = s1 + kr1*MOLECON(NOx, NO);

    /* determine O concentration (partial equilibrium)*/
    o_eq = ARRH(NOx, K_O)*sqrt(MOLECON(NOx, O2));

    /* calculate NO rate */
    s2 = 2.*o_eq;
    /* forward rate... */
    rf = s2*(kf1*MOLECON(NOx, N2))*s1/s3;
    /* reverse rate... */
    rr = -s2*kr1*kr2*pow(MOLECON(NOx, NO), 2.0)/s3;
    /* rates have units mole/m^3/s */
    NOX_FRATE(NOx) = rf;
    NOX_RRATE(NOx) = rr;
}
```

A number of Fluent-provided macros can be used in the calculation of pollutant rate using user-defined functions. The macros listed below are defined in the header file sg_nox.h which is included in the udf.h file. The variable NOx indicates a pointer to the NOx_Parameter structure.

- NOX_EQN(NOx) returns the index of the pollutant equation currently being solved. The indices are EQ_NO for NO, EQ_HCN for HCN, and EQ_NH3 for NH_3.
- MOLECON(NOx, SPE) returns the molar concentration of a species specified by SPE, which must be replaced by one of the following identifiers: FUEL, O_2, O, OH, H_2O, N_2, N, CH, CH_2, CH_3, NO, HCN, NH_3. Identifier FUEL represents the fuel species as specified in the **Fuel Species** drop-down list under **Prompt NO Parameters** in the **NOx Model** panel.
- NULLIDX(NOx, SPE) returns TRUE if the species specified by SPE does not exist in the **FLUENT** case (i.e., in the **Species** panel).
- ARRH(NOx, K) returns the Arrhenius rate calculated from the constants specified by K. K is defined using the Rate_Const data type and has three elements - *A*, *B*, and *C*. The Arrhenius rate is given in the form of

  $$R = AT^{B} \exp(-C/T)$$

  where *T* is the temperature.

- NOX_FRATE(NOx) is used to return the production rate of the pollutant species being solved.

- NOX_RRATE(NOx) is used to return the reduction rate of the pollutant species being solved.

Activating a NOx Rate UDF in FLUENT

Once you have compiled and linked your NOx rate UDF, you can activate it by selecting it in the **NOx Model** panel in **FLUENT**. See Section 8.2.3 for more details.

4.3.5 DEFINE_PROFILE

功能和使用方法的介绍

你可以使用 DEFINE_PROFILE macro to define a custom boundary profile that varies as a function of spatial coordinates or time. Some of the variables you can customize at a boundary are

- velocity, pressure, temperature, turbulent kinetic energy, turbulent dissipation rate
- volume fraction (multiphase)
- species mass fraction (species transport)
- wall thermal conditions
- wall stress conditions

Macro:	DEFINE_PROFILE (name, t, i)
Argument types:	Thread *t
	int i
Function returns:	void

There are three arguments to DEFINE_PROFILE: name, t, and i. name is the name of the UDF, specified by you. 当你的 UDF 编译和连接时，你为函数所选择的名字会在 FLUENT 图形用户界面中变得可见，且可被选择。 t and i are variables that are passed by the **FLUENT** solver to your UDF.

t is a pointer to the thread on which the boundary condition is to be applied. i is an index that identifies the variable that is to be defined. i is set when you associate (hook) the UDF with a variable in a boundary condition panel through the graphical user interface. This index is subsequently passed to your UDF by the **FLUENT** solver, so that your function knows which variable to operate on.

While DEFINE_PROFILE is usually used to specify a profile condition on a boundary face zone, it can also be used to specify, or fix, flow variables that are held constant during computation in a cell zone. See Section 6.26 of the User's Guide for more information on fixing values in a cell zone boundary condition. The arguments of the macro will change accordingly. There are three arguments to DEFINE_PROFILE for a cell zone: name, thread, and np. name is the name of the UDF, specified by you. thread and np are variables that are passed by the **FLUENT** solver to your UDF.

Note that unlike source term and property UDFs, profile UDFs (defined using DEFINE_PROFILE) are *not* called by **FLUENT** from within a loop on threads in the boundary zone. The solver only passes the DEFINE_PROFILE macro the pointer to the thread associated with the boundary zone. Your UDF will need to do the work of looping over all of the faces in the thread, compute the face value for the boundary variable, and then store the value in memory. Fluent has provided you with a face looping macro to loop over all faces in a thread (begin_f_loop...). See Chapter 6 for details about face looping macro utilities.

F_PROFILE is typically used along with DEFINE_PROFILE, and is a predefined macro provided by Fluent. F_PROFILE stores a boundary condition in memory for a given face and index and is nested within the face loop in the 例子：s below. It is important to note that the index i that is an argument to DEFINE_PROFILE is the same argument to F_PROFILE. F_PROFILE uses the thread pointer t, face identifier f, and index i to set the appropriate boundary face value in memory. See Section 6.4 for a description of F_PROFILE.

例子： 1

下面的 UDF 名字为 pressure_profile, generates a parabolic pressure profile according to the equation

$$p(y) = 1.1 \times 10^5 - 0.1 \times 10^5 \left(\frac{y}{0.0745}\right)^2$$

Note that this UDF assumes that the grid is generated such that the origin is at the geometric center of the boundary zone to which the UDF is to be applied. *y* is 0.0 at the center of the inlet and extends to ± 0.0745 at the top and bottom of the inlet. This function can be executed as an interpreted or compiled UDF in **FLUENT**.

```
/**********************************************************************
***/
```

```
/* UDF for specifying steady-state parabolic pressure profile boundary */
/* profile for a turbine vane                                          */
/**********************************************************************
***/

#include "udf.h"

DEFINE_PROFILE(pressure_profile, t, i)
{
   real x[ND_ND];            /* this will hold the position vector */
   real y;
   face_t f;

   begin_f_loop(f, t)
      {
         F_CENTROID(x,f,t);
         y = x[1];
         F_PROFILE(f, t, i) = 1.1e5 - y*y/(.0745*.0745)*0.1e5;
      }
   end_f_loop(f, t)
}
```

The function named pressure_profile has two arguments: t and i. t is a pointer to the face's thread, and i is an integer that is a numerical label for the variable being set within each loop.

Within the function body, variable f is declared as a face. A one-dimensional array x and variable y are declared as real data types. Following the variable declarations, a looping macro is used to loop over each face in the zone to create a profile, or an array of data. Within each loop, F_CENTROID returns the value of the face centroid (array x) for the face with index f that is on the thread pointed to by t. The *y* coordinate stored in x[1] is assigned to variable y, and is then used to calculate the pressure. This value is then assigned to F_PROFILE, which uses the integer i (passed to it by the solver, based on your selection of the UDF as the boundary condition for pressure in the **Pressure Inlet** panel) to set the pressure face value in memory.

例子： 2

在下面的例子中，：, DEFINE_PROFILE is used to generate profiles for the *x* velocity, turbulent kinetic energy, and dissipation rate, respectively, for a 2D fully-developed duct flow. Three separate UDFs named x_velocity, k_profile, and dissip_profile are defined. These functions are concatenated in a single C source file which can be executed as interpreted or compiled in **FLUENT**.

The 1/7th power law is used to specify the *x* velocity component:

$$v_x = v_{x,\text{free}} \left(\frac{y}{\delta}\right)^{1/7}$$

A fully-developed profile occurs when δ is one-half the duct height. In this 例子：, the mean *x* velocity is prescribed and the peak (free-stream) velocity is determined by averaging across the channel.

The turbulent kinetic energy is assumed to vary linearly from a near-wall value of

$$k_{\text{nw}} = \frac{u_\tau^2}{\sqrt{C_\mu}}$$

to a free-stream value of

$$k_{\text{inf}} = 0.002 u_{\text{free}}^2$$

The dissipation rate is given by

$$\epsilon = \frac{C_\mu^{3/4} (k^{3/2})}{\ell}$$

where the mixing length ℓ is the minimum of κy and 0.085 δ. (κ is the von Karman constant = 0.41.)

The friction velocity and wall shear take the forms

$$u_\tau = \sqrt{\tau_w / \rho}$$

$$\tau_w = \frac{f \rho u_{\text{free}}^2}{2}$$

The friction factor is estimated from the Blasius equation:

$$f = 0.045 \left(\frac{u_{\text{free}} \delta}{\nu} \right)^{-1/4}$$

```
/*******************************************************************
**/
/*      Concatenated UDFs for fully-developed turbulent inlet profiles    */
/*******************************************************************
**/

#include "udf.h"

#define YMIN 0.0                        /* constants    */
#define YMAX 0.4064
#define UMEAN 1.0
#define B 1./7.
#define DELOVRH 0.5
#define VISC 1.7894e-05
#define CMU 0.09
#define VKC 0.41


/*    profile for x-velocity        */


DEFINE_PROFILE(x_velocity, t, i)
{
   real y, del, h, x[ND_ND], ufree;          /* variable declarations */
   face_t f;

   h = YMAX - YMIN;
   del = DELOVRH*h;
   ufree = UMEAN*(B+1.);

   begin_f_loop(f, t)
      {
         F_CENTROID(x,f,t);
         y = x[1];

         if (y <= del)
              F_PROFILE(f,t,i) = ufree*pow(y/del,B);
         else
              F_PROFILE(f,t,i) = ufree*pow((h-y)/del,B);
```

```
      }
   end_f_loop(f, t)
}


/*    profile for kinetic energy    */


DEFINE_PROFILE(k_profile, t, i)
{
   real y, del, h, ufree, x[ND_ND];
   real ff, utau, knw, kinf;
   face_t f;

   h = YMAX - YMIN;
   del = DELOVRH*h;
   ufree = UMEAN*(B+1.);
   ff = 0.045/pow(ufree*del/VISC,0.25);
   utau=sqrt(ff*pow(ufree,2.)/2.0);
   knw=pow(utau,2.)/sqrt(CMU);
   kinf=0.002*pow(ufree,2.);


   begin_f_loop(f, t)
      {
         F_CENTROID(x,f,t);
         y=x[1];

         if (y <= del)
            F_PROFILE(f,t,i)=knw+y/del*(kinf-knw);
         else
            F_PROFILE(f,t,i)=knw+(h-y)/del*(kinf-knw);
      }
   end_f_loop(f, t)
}


/* profile for dissipation rate    */


DEFINE_PROFILE(dissip_profile, t, i)
{
   real y, x[ND_ND], del, h, ufree;
   real ff, utau, knw, kinf;
```

```
   real mix, kay;
   face_t f;

   h = YMAX - YMIN;
   del = DELOVRH*h;
   ufree = UMEAN*(B+1.);
   ff = 0.045/pow(ufree*del/VISC,0.25);
   utau=sqrt(ff*pow(ufree,2.)/2.0);
   knw=pow(utau,2.)/sqrt(CMU);
   kinf=0.002*pow(ufree,2.);

   begin_f_loop(f, t)
     {
       F_CENTROID(x,f,t);
       y=x[1];

       if (y <= del)
         kay=knw+y/del*(kinf-knw);
       else
         kay=knw+(h-y)/del*(kinf-knw);

       if (VKC*y < 0.085*del)
         mix = VKC*y;
       else
         mix = 0.085*del;

       F_PROFILE(f,t,i)=pow(CMU,0.75)*pow(kay,1.5)/mix;
     }
   end_f_loop(f,t)
}
```

例子： **3**

在下面的例子中：, DEFINE_PROFILE is used to fix flow variables that are held constant during computation in a cell zone. Three separate UDFs named fixed_u, fixed_v, and fixed_ke are defined in a single C source file. They specify fixed velocities that simulate the transient startup of an impeller in an impeller-driven mixing tank. The physical impeller is simulated by fixing the velocities and turbulence quantities using the fix option. See Section 6.26 of the User's Guide for more information on fixing variables.

```
/******************************************************************
***/
/* Concatenated UDFs for simulating an impeller using fixed velocity     */
```

```
/*******************************************************************
***/

#include "udf.h"

#define FLUID_ID 1
#define ua1 -7.1357e-2
#define ua2 54.304
#define ua3 -3.1345e3
#define ua4 4.5578e4
#define ua5 -1.9664e5

#define va1 3.1131e-2
#define va2 -10.313
#define va3 9.5558e2
#define va4 -2.0051e4
#define va5 1.1856e5

#define ka1 2.2723e-2
#define ka2 6.7989
#define ka3 -424.18
#define ka4 9.4615e3
#define ka5 -7.7251e4
#define ka6 1.8410e5

#define da1 -6.5819e-2
#define da2 88.845
#define da3 -5.3731e3
#define da4 1.1643e5
#define da5 -9.1202e5
#define da6 1.9567e6




DEFINE_PROFILE(fixed_u, thread, np)
{
   cell_t c;
   real x[ND_ND];
   real r;

   begin_c_loop (c,thread)
      {
/* centroid is defined to specify position dependent profiles*/
```

```
            C_CENTROID(x,c,thread);
            r =x[1];
            F_PROFILE(c,thread,np) =
ua1+(ua2*r)+(ua3*r*r)+(ua4*r*r*r)+(ua5*r*r*r*r);
}
   end_c_loop (c,thread)
}


DEFINE_PROFILE(fixed_v, thread, np)
{
   cell_t c;
   real x[ND_ND];
   real r;

   begin_c_loop (c,thread)
      {
/* centroid is defined to specify position dependent profiles*/

            C_CENTROID(x,c,thread);
            r =x[1];
            F_PROFILE(c,thread,np) =
va1+(va2*r)+(va3*r*r)+(va4*r*r*r)+(va5*r*r*r*r);
}
   end_c_loop (c,thread)
}



DEFINE_PROFILE(fixed_ke, thread, np)
{
   cell_t c;
   real x[ND_ND];
   real r;

   begin_c_loop (c,thread)
      {
/* centroid is defined to specify position dependent profiles*/

            C_CENTROID(x,c,thread);
            r =x[1];
            F_PROFILE(c,thread,np) =
ka1+(ka2*r)+(ka3*r*r)+(ka4*r*r*r)+(ka5*r*r*r*r)+(ka6*r*r*r*r*r);
```

```
        }
    end_c_loop (c,thread)
}
```

Activating a Profile UDF in FLUENT

Once you have compiled and linked your profile UDF, you can activate it by selecting it in the appropriate boundary condition panel in **FLUENT** (e.g., the **Velocity Inlet** panel). See Section 8.2.1 for more details.

4.3.6 DEFINE_PROPERTY

功能和使用方法的介绍

你可以使用 DEFINE_PROPERTY macro to specify a custom material property in **FLUENT**. Some of the properties you can customize are

- density (as a function of temperature only)
- viscosity
- thermal conductivity
- mass diffusivity
- absorption and scattering coefficients
- laminar flow speed
- rate of strain
- particle or droplet diameter (multiphase mixture model)

!! UDFs cannot be used to define specific heat properties; specific heat data can only be accessed, not modified in **FLUENT**.

Macro:	DEFINE_PROPERTY (name, c, t)
Argument types:	cell_t c
	Thread *t
Function returns:	real

There are three arguments to DEFINE_PROPERTY: name, c, and t. name is the name of the UDF, specified by you. 当你的 UDF 编译和连接时，你为函数所选择的名字会在 FLUENT 图形用户界面中变得可见，且可被选择。 c and t are variables that are passed by the **FLUENT** solver to your UDF.

c is an index that identifies a cell within the given thread. t is a pointer to the thread on which the material property definition is to be applied.

Note that like source term UDFs, property UDFs (defined using DEFINE_PROPERTY) are called by **FLUENT** from within a loop on cell threads. The solver passes to a DEFINE_PROPERTY UDF all of the necessary variables it needs to define a custom material, since properties are assigned on a cell basis. Consequently, your UDF will *not* need to loop over cells in a zone since **FLUENT** is already doing it. Your UDF will only be required to compute the property for a single cell and return the real value to the solver, as shown in the 例子： below.

例子：

下面的 UDF 名字为 cell_viscosity, generates a variable viscosity profile to simulate solidification. The function is called for every cell in the zone. The viscosity in the warm (T > 288 K) fluid has a molecular value for the liquid (5.5 $\times 10^{-3}$ kg/m-s), while the viscosity for the cooler region (T < 286 K) has a much larger value (1.0 kg/m-s). In the intermediate temperature range (286 K $\leq T \leq$ 288 K), the viscosity follows a linear profile that extends between the two values given above:

$$\mu = 143.2135 - 0.49725T \qquad (4.3.8)$$

This model is based on the assumption that as the liquid cools and rapidly becomes more viscous, its velocity will decrease, thereby simulating solidification. Here, no correction is made for the energy field to include the latent heat of freezing. The UDF can be executed as interpreted or compiled in **FLUENT**.

```
/******************************************************************
*/
/* UDF that simulates solidification by specifying a temperature-     */
/* dependent viscosity property                                        */
/******************************************************************
*/

#include "udf.h"

DEFINE_PROPERTY(cell_viscosity, c, t)
{
  real mu_lam;
  real temp = C_T(c, t);
```

```
   if (temp > 288.)
      mu_lam = 5.5e-3;
   else if (temp > 286.)
      mu_lam = 143.2135 - 0.49725 * temp;
   else
      mu_lam = 1.;

   return mu_lam;
}
```

The function cell_viscosity is defined on a cell. Two real variables are introduced: temp, the value of C_T(c,t), and mu_lam, the laminar viscosity computed by the function. The value of the temperature is checked, and based upon the range into which it falls, the appropriate value of mu_lam is computed. At the end of the function, the computed value for the viscosity (mu_lam) is returned to the solver.

Activating a Material Property UDF in FLUENT

Once you have compiled and linked your property UDF, you can activate it by selecting it in the **Materials** panel in **FLUENT**. See Section 8.2.4 for more details.

4.3.7 DEFINE_SCAT_PHASE_FUNC

功能和使用方法的介绍

你可以使用 DEFINE_SCAT_PHASE_FUNC macro to define the radiation scattering phase function for the discrete ordinates model. The function computes two values: the fraction of radiation energy scattered from direction *i* to direction *j*, and the forward scattering factor.

Macro:	DEFINE_SCAT_PHASE_FUNC (name, cosine, f)
Argument types:	real cosine
	real *f
Function returns:	real

There are three arguments to DEFINE_SCAT_PHASE_FUNC: name, cosine, and f. name is the name of the UDF, specified by you. 当你的 UDF 编译和连接时，你为函数所选择的名字会在 FLUENT 图形用户界面中变得可见，且可被选择。cosine and f are variables that are passed by the **FLUENT** solver to your UDF.

cosine is a real variable that is the cosine of the angle between the two directions. f is a real pointer that points to the location in memory where the real forward scattering factor is stored. The solver computes and stores a scattering matrix for each material by calling this function for each unique pair of discrete ordinates. Your UDF will need to return the fraction of radiation energy scattered from direction *i* to direction *j* to the solver, as shown in the 例子： below. The forward scattering factor is stored in the real variable that is referenced by the pointer f.

例子：

在下面的例子中，：, a number of UDFs are concatenated in a single C source file. These UDFs implement backward and forward scattering phase functions that are cited by Jendoubi et al. [1]. The source code can be executed as interpreted or compiled in **FLUENT**.

```
/*******************************************************************/
/*    UDFs that implement backward and forward scattering          */
/*    phase functions as cited by Jendoubi et. al.                 */
/*******************************************************************/

#include "udf.h"

DEFINE_SCAT_PHASE_FUNC(ScatPhiB2, c, fsf)
{
   real phi=0;
   *fsf = 0;
   phi = 1.0 - 1.2*c + 0.25*(3*c*c-1);
   return (phi);
}

DEFINE_SCAT_PHASE_FUNC(ScatPhiB1, c,fsf)
{
   real phi=0;
   *fsf = 0;
   phi = 1.0 - 0.56524*c + 0.29783*0.5*(3*c*c-1) +
      0.08571*0.5*(5*c*c*c-3*c) + 0.01003/8*(35*c*c*c*c-30*c*c+3) +
      0.00063/8*(63*c*c*c*c*c-70*c*c*c+15*c);
   return (phi);
}

DEFINE_SCAT_PHASE_FUNC(ScatPhiF3, c, fsf)
{
   real phi=0;
   *fsf = 0;
   phi = 1.0 + 1.2*c + 0.25*(3*c*c-1);
```

```
   return (phi);
}
DEFINE_SCAT_PHASE_FUNC(ScatPhiF2, c, fsf)
{
   real phi=0;
   real coeffs[9]={1,2.00917,1.56339,0.67407,0.22215,0.04725,
                          0.00671,0.00068,0.00005};
   real P[9];
   int i;
   *fsf = 0;
   P[0] = 1;
   P[1] = c;
   phi = P[0]*coeffs[0] + P[1]*coeffs[1];
   for(i=1;i<7;i++)
      {
         P[i+1] = 1/(i+1.0)*((2*i+1)*c*P[i] - i*P[i-1]);
         phi += coeffs[i+1]*P[i+1];
      }
   return (phi);
}

DEFINE_SCAT_PHASE_FUNC(ScatIso, c, fsf)
{
   *fsf=0;
   return (1.0);
}
```

Activating a Radiation Scattering Phase UDF in FLUENT

Once you have compiled and linked your scattering phase UDF, you can activate it by selecting it in the **Materials** panel in **FLUENT**. See Section 8.2.4 for more details.

4.3.8 DEFINE_SOURCE

功能和使用方法的介绍

你可以使用 DEFINE_SOURCE macro to define custom source terms for the different types of solved transport equations in **FLUENT** (except the discrete ordinates radiation model) including:

- continuity
- momentum
- *k*, ϵ
- energy (also for solid zones)
- species mass fractions

- user-defined scalar (UDS) transport

Macro:	DEFINE_SOURCE (name, c, t, dS, eqn)
Argument types:	cell_t c
	Thread *t
	real dS[]
	int eqn
Function returns:	real

There are five arguments to DEFINE_SOURCE: name, c, t, dS, and eqn. name is the name of the UDF, specified by you. 当你的 UDF 编译和连接时，你为函数所选择的名字会在 FLUENT 图形用户界面中变得可见，且可被选择。c, t, dS, and eqn are variables that are passed by the **FLUENT** solver to your UDF.

c is the index of a cell on the thread pointed to by t. This is the cell in which the source term is to be applied. Array dS specifies the derivative of the source term with respect to the dependent variable of the transport equation. These derivatives may be used to linearize the source term if they enhance the stability of the solver. To see this, note that the source term can be expressed, in general, as Equation 4.3-10, where ϕ is the dependent variable, *A* is the explicit part of the source term, and $B\phi$ is the implicit part.

$$S_\phi = A + B\phi \qquad (4.3.9)$$

Specifying a value for *B* in Equation 4.3-10 can enhance the stability of the solution and help convergence rates due to the increase in diagonal terms on the solution matrix. **FLUENT** automatically determines if the value of *B* that is given by the user will aid stability. If it does, then **FLUENT** will define *A* as $S^* - (\partial S/\partial\phi)^*\phi^*$, and *B* as $(\partial S/\partial\phi)^*$. If not, the source term is handled explicitly. Your UDF will need to return the real value of the total source term to the solver, but you have the choice of setting the implicit term dS[eqn] to $dS/d\phi$, or forcing the explicit solution of the source term by setting it equal to 0.0. See Section 1.5.1 for a list of the transport

equations for which source terms can be applied and the corresponding ϕ variable that is used to derive the $dS/d\phi$ term (dS[eqn]).

Note that like property UDFs, source term UDFs (defined using DEFINE_SOURCE) are called by **FLUENT** from within a loop on cell threads. The solver passes to the DEFINE_SOURCE term UDF all the necessary variables it needs to define a custom source term, since source terms are solved on a cell basis. Consequently, your UDF will *not* need to loop over cells in the thread since **FLUENT** is already doing it. Your UDF will only be required to compute the source term for a single cell and return the real value to the solver. Note that the units on all source terms are of the form generation-rate/volume. For 例子：, a source term for the continuity equation would have units of kg/m ^{3}-s.

例子：

下面的 UDF 名字为 xmom_source, is used to add source terms in **FLUENT**. It can be executed as an interpreted or compiled UDF. The function generates an *x*-momentum source term that varies with *y* position as

$$\text{source} = -0.5 C_2 \rho y |v_x| v_x$$

Suppose

$$\text{source} = S = -A |v_x| v_x$$

where

$$A = 0.5 C_2 \rho y$$

Then

$$\frac{dS}{dv_x} = -A|v_x| - Av_x \frac{d}{dv_x}(|v_x|)$$

The source term returned is

$$\text{source} = -A|v_x|v_x$$

and the derivative of the source term with respect to v_x (true for both positive and negative values of v_x) is

$$\frac{dS}{dv_x} = -2A|v_x|$$

```
/******************************************************************/
/* UDF for specifying an x-momentum source term in a spatially          */
/* dependent porous media                                               */
/******************************************************************/

#include "udf.h"

#define C2 100.0

DEFINE_SOURCE(xmom_source, c, t, dS, eqn)
{
   real x[ND_ND];
   real con, source;

   C_CENTROID(x, c, t);
   con = C2*0.5*C_R(c, t)*x[1];
```

```
    source = -con*fabs(C_U(c, t))*C_U(c, t);
    dS[eqn] = -2.*con*fabs(C_U(c, t));

    return source;
}
```

Activating a Source Term UDF in FLUENT

Once you have compiled and linked your source term UDF, you can activate it by selecting it in the **Fluid** or **Solid** panel in **FLUENT**. See Section 8.2.7 for more details.

4.3.9 DEFINE_SR_RATE

功能和使用方法的介绍

你可以使用 DEFINE_SR_RATE macro to customize a surface reaction rate.

Macro:	DEFINE_SR_RATE (name, f, t, r, my, yi, rr)
Argument types:	face_t f
	Thread *t
	Reaction *r
	real *mw
	real *yi
	real *rr
Function returns:	void

There are seven arguments to DEFINE_SR_RATE: name, f, t, r, my, yi, and rr. name is the name of the UDF, specified by you. 当你的 UDF 编译和连接时，你为函数所选择的名字会在 FLUENT 图形用户界面中变得可见，且可被选择。f, t, r, my, and yi are variables that are passed by the **FLUENT** solver to your UDF.

f is an index that identifies a face within the given thread. t is a pointer to the thread on which the surface rate reaction is to be applied. r is a pointer to the data structure for the reaction. mw is a pointer to a real array containing the species molecular weights, and yi is a pointer to a real array containing the species mass fractions.

Your UDF will need to set the reaction rate to the value referenced by the real pointer rr as shown in the 例子： below.

例子：

下面编译的 UDF 名字为 arrhenius, defines a custom surface reaction rate in **FLUENT**.

```
/*********************************************************************/
/* UDF for specifying a custom surface reaction rate                  *
/*********************************************************************/

#include "udf.h"

/* ARRHENIUS CONSTANTS */
#define PRE_EXP 1e+15
#define ACTIVE 1e+08
#define BETA 0.0

real arrhenius_rate(real temp)
{
  return
PRE_EXP*pow(temp,BETA)*exp(-ACTIVE/(UNIVERSAL_GAS_CONSTANT*te
mp));
}

/* Species numbers. Must match order in Fluent panel
*/
#define HF    0
#define WF6 1
#define H2O 2
#define NUM_SPECS 3

/* Reaction Exponents */
#define HF_EXP    2.0
#define WF6_EXP 0.0
#define H2O_EXP 0.0

#define MW_H2 2.0
#define STOIC_H2 3.0
```

```
/* Reaction Rate Routine that is used in both UDFs */
real reaction_rate(cell_t c, Thread *cthread,real
mw[],real yi[])
{
  real concenHF = C_R(c, cthread)*yi[HF]/mw[HF];

  return arrhenius_rate(C_T(c,
cthread))*pow(concenHF,HF_EXP);
}
DEFINE_SR_RATE(arrhenius,f,fthread,r,mw,yi,rr)
{
  *rr =
reaction_rate(F_C0(f,fthread),F_C0_THREAD(f,fthread),mw,yi);
}

real contact_area(cell_t c, Thread *t, int s_id, int *n)
{
  int i = 0;
  real area = 0.0, A[ND_ND];

  *n = 0;
  c_face_loop(c,t,i)
  {
    if(THREAD_ID(C_FACE_THREAD(c,t,i)) == s_id)
    {
      (*n)++;
      F_AREA(A, C_FACE(c,t,i), C_FACE_THREAD(c,t,i));
      area += NV_MAG(A);
    }
  }

  return area;
}
```

Activating a Surface Reaction Rate UDF in FLUENT

Once you have compiled and linked the source code for a custom surface reaction rate UDF, you can activate it in the **User-Defined Function Hooks** panel in **FLUENT**. See Section 8.2.6 for more details.

4.3.10 DEFINE_TURB_PREMIX_SOURCE

功能和使用方法的介绍

你可以使用 DEFINE_TURB_PREMIX_SOURCE macro to customize the turbulent flame speed and source term in the premixed combustion model (see Chapter 15 of the User's Guide) and the partially premixed combustion model (see Chapter 16).

Macro:	DEFINE_TURB_PREMIX_SOURCE (name, c, t,
	turb_flame_speed, source)
Argument types:	cell_t c
	Thread *t
	real *turb_flame_speed
	real *source
Function returns:	void

There are five arguments to DEFINE_TURB_PREMIX_SOURCE: name, c, t, turb_flame_speed, and source. name is the name of the UDF, specified by you. 当你的 UDF 编译和连接时，你为函数所选择的名字会在 FLUENT 图形用户界面中变得可见，且可被选择。 c, t, turb_flame_speed, and source are variables that are passed by the **FLUENT** solver to your UDF.

t is a pointer to the thread on which the turbulent premixed source term is to be applied. c is an index that identifies a cell within the given thread. turb_flame_speed is a real pointer to the turbulent flame speed, and source is a real pointer to the reaction progress source term.

Your UDF will need to set the turbulent flame speed to the value referenced by the turb_flame_speed pointer. It will also need to set the source term to the value referenced by the source pointer. This is demonstrated in the 例子： below.

例子：

下面的 UDF 名字为 turb_flame_src, specifies a custom turbulent flame speed and source term in the premixed combustion model. It can be executed as a compiled UDF in **FLUENT**.

In the standard premixed combustion model in **FLUENT**, the mean reaction rate of the progress variable (that is, the source term) is modeled as

$$\rho S_c = \rho_u U_t |\nabla c| \tag{4.3.10}$$

where c is the mean reaction progress variable, ρ is the density, and U_t is the turbulent flame speed.

In the UDF 例子：, the turbulent flame speed is modeled as

$$U_t = U_l \sqrt{1 + (u'/U_l)^2} \tag{4.3.11}$$

where U_l is the laminar flame speed and u' is the turbulent fluctuation. Note that the partially premixed combustion model is assumed to be enabled (see Chapter 16 of the User's Guide), so that the unburned density and laminar flame speed are available as polynomials. See Chapter 6 for details on the NULLP, THREAD_STORAGE, and SV_VARS utilities.

```
/*******************************************************************/
/*    UDF that specifies a custom turbulent flame speed and source       */
/*    for the premixed combustion model                                  */
/*******************************************************************/

#include "udf.h"
#include "sg_pdf.h" /* not included in udf.h so must include here    */

DEFINE_TURB_PREMIX_SOURCE(turb_flame_src,  c,  t,  turb_flame_speed, source)
{
   real up = TRB_VEL_SCAL(c,t);
   real ut, ul, grad_c, rho_u, DV[ND_ND];

   ul       = Ul_par_premix(C_FMEAN(c,t));
   rho_u = rho_par_premix(C_FMEAN(c,t));
```

```
    if( NNULLP(THREAD_STORAGE(t,SV_PREMIXC_G)) )
      {
        NV_V (DV, =, C_STORAGE_R_NV(c,t,SV_PREMIXC_G));
        grad_c = sqrt( NV_DOT(DV, DV) );
      }

    ut = ul*sqrt( 1. + SQR(up/ul) );

    *turb_flame_speed = ut;
    *source = rho_u*ut*grad_c;
}
```

Activating a Turbulent Premixed Source UDF in FLUENT

Once you have compiled and linked the source code for a custom turbulent premixed source UDF, you can activate it in the **User-Defined Function Hooks** panel in **FLUENT**. See Section 8.2.5 for more details.

4.3.11 DEFINE_TURBULENT_VISCOSITY

功能和使用方法的介绍

你可以使用 DEFINE_TURBULENT_VISOCITY macro to define a custom turbulent viscosity for the Spalart-Allmaras, *k*- ϵ, *k*- ω, and LES turbulence models.

Macro:	DEFINE_TURBULENT_VISCOSITY (name, c, t)
Argument types:	cell_t c Thread *t
Function returns:	real

There are three arguments to DEFINE_TURBULENT_VISCOSITY: name, c, and t. name is the name of the UDF, specified by you. 当你的 UDF 编译和连接时，你为函数所选择的名字会在 FLUENT 图形用户界面中变得可见，且可被选择。 c and t are variables that are passed by the **FLUENT** solver to your UDF.

t is a pointer to the thread on which the turbulent viscosity is to be applied. c is an index that identifies a cell within the given thread. Your UDF will need to return the real value of the turbulent viscosity to the solver, as shown in the 例子： below. Note that the value of M_keCmu in the 例子： is defined through the graphical user interface, but made accessible to all UDFs.

例子：

下面的 UDF 名字为 user_mu_t, defines a custom turbulent viscosity for the standard *k*- ϵ turbulence model. This function can be executed as an interpreted or compiled UDF in **FLUENT**.

```
/**********************************************************************/
/*    UDF that specifies a custom turbulent viscosity for standard         */
/*          k-epsilon     formulation     using     DEFINE_TURBULENT_VISCOSITY
*/
/**********************************************************************/

#include "udf.h"

DEFINE_TURBULENT_VISCOSITY(user_mu_t, c, t)
{
   real mu_t;
   real rho = C_R(c,t);
   real k      = C_K(c,t);
   real d      = C_D(c,t);

   mu_t = M_keCmu*rho*SQR(k)/d;

   return mu_t;
}
```

Activating a Turbulent Viscosity UDF in FLUENT

Once you have compiled and linked the source code for a custom turbulent viscosity UDF, you can activate it in the **Viscous Model** panel in **FLUENT**. See Section 8.2.9 for more details.

4.3.12 DEFINE_UDS_FLUX

功能和使用方法的介绍

你可以使用 DEFINE_UDS_FLUX macro when you want to customize how the advective flux term is computed in your user-defined scalar (UDS) transport equation. Details on setting up and solving UDS transport equations are provided in Chapter 9.

Macro:	DEFINE_UDS_FLUX (name, f, t, i)
Argument types:	face_t f
	Thread *t
	int i
Function returns:	real

There are four arguments to DEFINE_UDS_FLUX: name, f, t, and i. name is the name of the UDF, specified by you. 当你的 UDF 编译和连接时，你为函数所选择的名字会在 FLUENT 图形用户界面中变得可见，且可被选择。 f, t, and i are variables that are passed by the **FLUENT** solver to your UDF.

t is a pointer to the thread on which the user-defined scalar flux is to be applied. f is an index that identifies a face within the given thread. i is an index that identifies the user-defined scalar. Your UDF will need to return the real value of the mass flow rate through the given face, as shown in the 例子： below.

The advection term in the differential transport equation has the following most general form:

$$\nabla \cdot \vec{\psi}\phi \qquad (4.3.12)$$

where ϕ is the user-defined scalar conservation quantity and $\vec{\psi}$ is a vector field. In the default advection term, $\vec{\psi}$ is, by default, the product of the scalar density and the velocity vector:

$$\vec{\psi}_{\mathrm{default}} = \rho\vec{v} \qquad (4.3.13)$$

To define the advection term in Equation 4.3-13 using DEFINE_UDS_FLUX, your UDF needs to return the scalar value $\vec{\psi}\cdot\vec{A}$ to **FLUENT**, where $\vec{\psi}$ is the same as defined in Equation 4.3-13 and $\vec{A}$ is the face normal vector of the face.

You will need to compute $\vec{\psi}$ in your UDF using, for 例子：, predefined macros for velocity vector and scalar density that Fluent has provided (see Chapter 6 for more details) or using your own prescription. The first case is illustrated in the sample C source code, shown below.

!! Note that if more than one scalar is being solved, you can use a conditional if statement in your UDF to define a different flux function for each i. i = 0 is associated with scalar-0 (the first scalar equation being solved).

!! Note also that $\vec{\psi}\cdot\vec{A}$ must have units of mass flow rate in SI (i.e., kg/s).

```
/*******************************************************************
*/
/*    sample C source code that computes dot product of psi and A            */
/*    Note that this is not a complete C function                             */
/*******************************************************************
*/


real NV_VEC(psi), NV_VEC(A);             /* declaring vectors psi and A        */
NV_D(psi,    =, F_U(f,t), F_V(f,t), F_W(f,t));      /* defining psi in        */
                                                   /* terms of velocity field          */
NV_S(psi, *=, F_R(f,t))      /*    multiplying density to get psi vector    */
F_AREA(A, f, t)                    /* face normal vector returned from F_AREA */
return NV_DOT(psi, A);        /* dot product of the two returned                 */
```

Additionally, since most quantities in **FLUENT** are not allocated in memory for interior faces, only for boundary faces (e.g., wall zones), your UDF will also need to calculate interior face values from the cell values of adjacent cells. This is most easily done using the arithmetic mean method. Vector arithmetic can be coded in C using the NV_ and ND_ macros that Fluent has provided (see Chapter 6 for more details).

Note that if you had to implement the default advection term in a UDF without the fluid density in the definition of ψ (see above), you could simply put the following line in your DEFINE_UDS_FLUX UDF:

return F_FLUX(f,t) / rho;

where the denominator ρ can be determined by averaging the adjacent cell's density values C_R(F_C0(f,t),F_C0_THREAD(f,t)) and C_R(F_C1(f,t),F_C1_THREAD(f,t)).

例子：

下面的 UDF 名字为 my_uds_flux, returns the mass flow rate through the given face. This value is usually available within **FLUENT** through the F_FLUX(f,t) macro. The flux is positive when it goes from cell c0 to cell c1. The face flux normal vector always points from c0 to c1. Note that the face area normal vector (returned by F_AREA also points outwards (out of the domain) for boundary faces. The UDF can be executed as a compiled UDF in **FLUENT**.

```
/*******************************************************************
**/
/*      UDF that implements a simplified advective term in the          */
/*      scalar transport equation                                        */
/*******************************************************************
**/

#include "udf.h"

DEFINE_UDS_FLUX(my_uds_flux, f, t, i)
{
   Thread *t0, *t1 = NULL;
   cell_t    c0,    c1 = -1;

   real NV_VEC(psi_vec), NV_VEC(A);

   /* neighboring cells of face f, and their (corresponding) threads */
```

```
    t0 = F_C0_THREAD(f,t);
    c0 = F_C0(f,t);

    if (NULL != F_C1_THREAD(f,t))
    /* Alternative: if (! BOUNDARY_FACE_THREAD_P(t)) */
      {
        t1 = F_C1_THREAD(f,t);
        c1 = F_C1(f,t);
      }
    else
      {
        t1 = NULL;
        c1 = -1;
      }

    /* If Face lies at domain boundary, use face values; */
    /* If Face lies IN the domain, use average of adjacent cells. */

    if (NULL == t1)
    /* Alternative: if (BOUNDARY_FACE_THREAD_P(t)) */
      {
        NV_D(psi_vec,    =, F_U(f,t), F_V(f,t), F_W(f,t));
        NV_S(psi_vec, *=, F_R(f,t));
      }
    else
      {
        NV_D(psi_vec,    =, C_U(c0,t0), C_V(c0,t0), C_W(c0,t0));
        NV_D(psi_vec, +=, C_U(c1,t1), C_V(c1,t1), C_W(c1,t1));
        NV_S(psi_vec, /=, 2.);          /* averaging. */
        NV_S(psi_vec, *=, (((C_R(c0,t0) + C_R(c1,t1)) / 2.)));
      }

    /* Now psi_vec contains our "psi" from above. */
    /* Next, get the face normal vector: */

    F_AREA(A, f, t);

    /* Finally, return the dot product of both.    */
    /* Fluent will multiply the returned value      */
    /* by phi_f (the scalar's value at the face) */
    /* to get the "complete" advective term...      */

    return NV_DOT(psi_vec, A);
}
```

Activating a User-Defined Scalar Flux UDF in FLUENT

Once you have compiled and linked the source code for a UDS flux UDF, you can activate it in **FLUENT** in the **User-Defined Scalars** panel. See Section 8.2.10 for more details.

4.3.13 DEFINE_UDS_UNSTEADY

功能和使用方法的介绍

你可以使用 DEFINE_UDS_UNSTEADY macro when you want to customize unsteady terms in your user-defined scalar (UDS) transport equations. Details on setting up and solving UDS transport equations are provided in Chapter 9.

Macro:	DEFINE_UDS_UNSTEADY (name, c, t, i, apu, su)
Argument types:	cell_t c
	Thread *t
	int i
	real *apu
	real *su
Function returns:	void

There are six arguments to DEFINE_UDS_UNSTEADY: name, c, t, i, apu, and su. name is the name of the UDF, specified by you. 当你的 UDF 编译和连接时，你为函数所选择的名字会在 FLUENT 图形用户界面中变得可见，且可被选择。 c, t, and i are variables that are passed by the **FLUENT** solver to your UDF.

t is a pointer to the thread on which the unsteady term for the UDS transport equation is to be applied. c is an index that identifies a cell within the given thread. i is an index that identifies the user-defined scalar for which the unsteady term is to be set.

Your UDF will need to set the values of the unsteady terms referenced by the real pointers apu and su to the central coefficient and source term, respectively. Note that

if more than one scalar is being solved, a conditional if statement can be used to define a different unsteady term for each i. i = 0 is associated with scalar-0 (the first scalar equation being solved).

The **FLUENT** solver expects that the transient term will be decomposed into a source term, su, and a central coefficient term, apu. These terms are included in the equation set in a similar manner to the way the explicit and implicit components of a source term might be handled. Hence, the unsteady term is moved to the RHS and discretized as follows:

$$\begin{aligned}\textbf{unsteady term} &= -\int \frac{\partial}{\partial t}(\rho\phi)\,dV \\ &\approx -\left[\frac{(\rho\phi)^n - (\rho\phi)^{n-1}}{\Delta t}\right]\cdot \Delta V \\ &= \underbrace{-\frac{\rho\Delta V}{\Delta t}}_{\text{apu}}\phi^n + \underbrace{\frac{\rho\Delta V}{\Delta t}\phi^{n-1}}_{\text{su}} \qquad (4.3.14)\end{aligned}$$

Note that Equation 4.3-15 shows how su and apu are defined.

例子：

下面的 UDF 名字为 uns_time, modifies user-defined scalar time derivatives using DEFINE_UDS_UNSTEADY. This function can be executed as an interpreted or compiled UDF in **FLUENT**.

```
/**********************************************************************
***/
/* UDF for specifying user-defined scalar time derivatives                */
/**********************************************************************
***/

#include "udf.h"

DEFINE_UDS_UNSTEADY(uns_time, c, t, i, apu, su)
{
   real physical_dt, vol, rho, phi_old;
```

```
   physical_dt = RP_Get_Real("physical-time-step");
   vol = C_VOLUME(c,t);

   rho = C_R_M1(c,t);
   *apu = -rho*vol / physical_dt;/*implicit part*/
   phi_old = C_STORAGE_R(c,t,SV_UDSI_M1(i));
   *su    = rho*vol*phi_old/physical_dt;/*explicit part*/
}
```

Activating an Unsteady UDS Term in FLUENT

Once you have compiled and linked the source code for an unsteady UDS term UDF, you can activate it in the **User-Defined Scalar** panel in **FLUENT**. See Section 8.2.11 for more details.

4.3.14 DEFINE_VR_RATE

功能和使用方法的介绍

你可以使用 DEFINE_VR_RATE macro when you want to define a custom volumetric reaction rate for a single reaction or for multiple reactions. During **FLUENT** execution, the same UDF is called sequentially for all reactions.

Macro: DEFINE_VR_RATE (name, c, t, r, mw, yi, rr, rr_t)

Argument types: cell_t c

Thread *t

Reaction *r

real *mw

real *yi

real *rr

real *rr_t

Function returns: void

There are eight arguments to DEFINE_VR_RATE: name, c, t, r, mw, yi, rr, and rr_t. name is the name of the UDF, specified by you. 当你的 UDF 编译和连接时，你为函数所选择的名字会在 FLUENT 图形用户界面中变得可见，且可被选择。 c, t, r, mw, yi, rr, and rr_t are variables that are passed by the **FLUENT** solver to your UDF.

t is a pointer to the thread on which the volumetric reaction rate is to be applied. c is an index that identifies a cell within the given thread. r is a pointer to the data structure that represents the current reaction. mw is a real pointer array of the species molecular weights, and yi is a real pointer array of the species mass fractions.

Your UDF will need to set the values referenced by the real pointers rr and rr_t to the laminar and turbulent reaction rates, respectively. rr and rr_t (defined by the UDF) are computed and the lower of the two values is used when the finite-rate/eddy-dissipation chemical reaction mechanism used. Note that rr and rr_t are conversion rates in kgmol/m ^{3}-s. These rates, when multiplied by the respective stoichiometric coefficients, yield the production/consumption rates of the individual chemical components. DEFINE_VR_RATE is called by **FLUENT** for every reaction in every single cell.

例子： 1

下面的 UDF 名字为 vol_reac_rate, specifies a volume reaction rate. The function can be executed as a compiled UDF in **FLUENT**.

```
/*******************************************************************
*/
/* UDF for specifying a volume reaction rate                          */
/* The basics of Fluent's calculation of reaction rates:   only an       */
/* Arrhenius ("finite rate") reaction rate is calculated                 */
/* from the inputs given by the user in the graphical user interface */
/*******************************************************************
*/

#include "udf.h"

DEFINE_VR_RATE(vol_reac_rate, c, t, r, wk, yk, rate, rr_t)
{
   real ci, prod;
   int i;

/* Calculate Arrhenius reaction rate    */

   prod = 1.;

   for(i = 0; i < r->n_reactants; i++)
```

```
    {
        ci          = C_R(c,t) * yk[r->reactant[i]] / wk[r->reactant[i]];
        prod    *= pow(ci, r->exp_reactant[i]);
    }
  *rate = r->A * exp( - r->E / (UNIVERSAL_GAS_CONSTANT * C_T(c,t))) *
                                                         pow(C_T(c,t), r->b) * prod;

  *rr_rate = *rate;

  /* No "return..;" value. */
}
```

例子： **2**

When multiple reactions are specified, the UDF is called several times in each cell. Different values are assigned to the pointer r, depending on which reaction the UDF is being called for. Therefore, you will need to determine which reaction is being called, and return the correct rates for that reaction. Reactions can be identified by their name through the r->name statement. To test whether a given reaction has the name reaction-1 for 例子：, 你可以使用 following C constuct:

```
if (!strcmp(r->name, "reaction-1"))
    {
      ....   /* r->name is identical to "reaction-1" ... */
    }
```

(Note that strcmp(r->name, "reaction-1") returns 0 which is equal to FALSE when the two strings are identical.)

It should be noted that DEFINE_VR_RATE only defines the reaction rate for a predefined stoichiometric equation (set in the **Reactions** panel) thus providing an alternative to the Arrhenius rate model. DEFINE_VR_RATE does not directly address the particular rate of species creation or depletion; this is done by the **FLUENT** solver using the reaction rate provided by your UDF.

The following is a simple template that shows how to use DEFINE_VR_RATE in connection with more than one user-specified reaction. Note that **FLUENT** always calculates the rr and rr_t reaction rates before the UDF is called. Consequently, the values that are calculated are available only in the given variables when the UDF is called.

```
/*****************************************************************
*/
/* Multiple reaction UDF that specifies different reaction rates          */
/* for different volumetric chemical reactions                                    */
```

```
/*******************************************************************
*/
#include "udf.h"

DEFINE_VR_RATE(myrate, c, t, r, mw, yi, rr, rr_t)
{
  /*If more than one reaction is defined, it is necessary to distinguish*/
  /* between these using the names of the reactions.                    */

          if (!strcmp(r->name, "reaction-1"))
             {
                /* Reaction 1 */
             }
          else if (!strcmp(r->name, "reaction-2"))
             {
                /* Reaction 2 */
             }
          else
             {
/*              CX_Message("Unknown Reaction\n"); */
             }
/*           CX_Message("Actual Reaction: %s\n",r->name); */

}
```

Activating a Volumetric Reaction Rate UDF in FLUENT

Once you have compiled and linked the source code for a custom volumetric reaction rate UDF, you can activate it in the **User-Defined Function Hooks** panel in **FLUENT**. See Section 8.2.6 for more details.

4.4 Multiphase DEFINE Macros

The following DEFINE macros are used only for multiphase applications. Table 4.4.1 provides a quick reference guide to the DEFINE macros, the functions they are used to define, and the panels where they are activated in **FLUENT**. Definitions of each DEFINE macro are listed in the udf.h header file. For your convenience, the definitions are also provided in Appendix A.

- DEFINE_CAVITATION_RATE (Section 4.4.1)
- DEFINE_EXCHANGE_PROPERTY (Section 4.4.2)
- DEFINE_VECTOR_EXCHANGE_PROPERTY (Section 4.4.3)

Table 4.4.1: Quick Reference Guide for Multiphase DEFINE Macros			
Function	DEFINE **Macro**	**Panel Activated In**	
cavitation rate	DEFINE_CAVITATION_RATE	**User-Defined Function Hooks**	
drag coefficient	DEFINE_EXCHANGE_PROPERTY	**Phase Interaction**	
lift coefficient	DEFINE_EXCHANGE_PROPERTY	**Phase Interaction**	
slip velocity	DEFINE_VECTOR_EXCHANGE_PROPERTY	**Phase Interaction**	

- 4.4.1 DEFINE_CAVITATION_RATE
- 4.4.2 DEFINE_EXCHANGE_PROPERTY
- 4.4.3 DEFINE_VECTOR_EXCHANGE_PROPERTY

4.4.1 DEFINE_CAVITATION_RATE

功能和使用方法的介绍

你可以使用 DEFINE_CAVITATION_RATE macro to model the creation of vapor due to pressure tension in a multiphase flow.

Macro:	DEFINE_CAVITATION_RATE (name, c, t, p, rhoV,
	rhoL, vofV, p_v, n_b, m_dot)
Argument types:	cell_t c
	Thread *t
	real *p
	real *rhoV
	real *rhoL

	real *vofV
	real *p_v
	real *n_b
	real *m_dot
Function returns:	void

There are ten arguments to DEFINE_CAVITATION_RATE: name, c, t, p, rhoV, rhoL, vofV, p_v, n_b, and m_dot. name is the name of the UDF, specified by you. 当你的 UDF 编译和连接时，你为函数所选择的名字会在 FLUENT 图形用户界面中变得可见，且可被选择。c, t, p, rhoV, rhoL, vofV, p_v, n_b, and m_dot are variables that are passed by the **FLUENT** solver to your UDF.

t is a pointer to the mixture-level thread. c is the index of a cell on the thread pointed to by t. The remaining arguments are real pointers to the following data: shared pressure (p), vapor density (rhoV), liquid density (rhoL), vapor volume fraction (vofV), vaporization pressure (p_v), number of bubbles per unit volume (n_b), and rate of vapor formation (m_dot) Your UDF will need to set the value referenced by the real pointer m_dot, to the cavitation rate.

例子：

下面的 UDF 名字为 cavitation_rate, is an 例子： of how to calculate the mass transfer between the liquid and vapor phases (cavitation rate) in a multiphase mixture. The function can be executed as an interpreted or compiled UDF.

```
/**********************************************************************
***/
/* UDF that computes the cavitation rate                                     */
/**********************************************************************
***/

#include "udf.h"

#define MIN_VOF 1.e-5
#define MAX_VOF 0.999999

DEFINE_CAVITATION_RATE(cavitation_rate, c, t, p,
                       rhoV, rhoL, vofV, p_v, n_b, m_dot)
{
   real p_vapor = *p_v;
   real n_bubbles = *n_b;
```

```
   real dp, vofM, radV;

   dp     = p_vapor - ABS_P( p[c], op_pres );
   vofM = MIN(MAX(MIN_VOF, vofV[c]),MAX_VOF);
   radV = pow(vofM/((1.-vofM)*4./3.*M_PI*n_bubbles), 1./3.);

   if (dp>0.)
         *m_dot = (1.-vofV[c]) * n_bubbles * 4. * M_PI *
                       radV * radV/(1.+n_bubbles*4./3.*M_PI*radV*radV*radV) *
                       sqrt(2.*ABS(dp)/(3.*rhoL[c]));
   else
      {
         *m_dot = - (1.-vofV[c]) * n_bubbles * 4. *M_PI *
                         radV*radV/(1.+n_bubbles*4./3.*M_PI*radV*radV*radV) *
                         sqrt(2.*ABS(dp)/(3.*rhoL[c]));
         if (vofV[c] <= MIN_VOF) *m_dot=0.;
      }

}
```

Activating a Cavitation Rate UDF in FLUENT

Once you have compiled and linked the source code for a custom cavitation rate UDF, you can activate it in the **User-Defined Function Hooks** panel in **FLUENT**. See Section 8.3.1 for more details.

4.4.2 DEFINE_EXCHANGE_PROPERTY

功能和使用方法的介绍

你可以使用 DEFINE_EXCHANGE_PROPERTY macro to specify custom lift and drag coefficients for the multiphase Eulerian model.

Macro:	DEFINE_EXCHANGE_PROPERTY (name, c, mixture_thread,
	second_column_phase_index, first_column_phase_index)
Argument types:	cell_t c
	Thread *mixture_thread
	int second_column_phase_index
	int first_column_phase_index
Function returns:	real

There are five arguments to DEFINE_EXCHANGE_PROPERTY: name, c, mixture_thread, second_column_phase_index, and first_column_phase_index. name is the name of the UDF, specified by you. 当你的 UDF 编译和连接时，你为函数所选择的名字会在 FLUENT 图形用户界面中变得可见，且可被选择。 c, mixture_thread, second_column_phase_index, and first_column_phase_index are variables that are passed by the **FLUENT** solver to your UDF.

mixture_thread is a pointer to the mixture-level thread. c is the index of a cell on the thread pointed to by mixture_thread. first_column_phase_index and second_column_phase_index are integer identifiers corresponding to the pair of phases in your multiphase flow that you are specifying a slip velocity for. The identifiers correspond to the phases that are selected in the **Phase Interaction** panel in the graphical user interface. An index of 0 corresponds to the primary phase, and is incremented by one for each secondary phase. Your UDF will need to return the real value of the lift or drag coefficient to the **FLUENT** solver.

例子：

下面的 UDF 名字为 custom_drag, can be used to customize the default Syamlal drag law in **FLUENT**. The default drag law uses 0.8 (for void <=0.85) and 2.65 (void >0.85) for bfac. This results in a minimum fluid velocity of 25 cm/s. The UDF modifies the drag law to result in a minimum fluid velocity of 8 cm/s, using 0.28 and 9.07 for the bfac parameters.

```
/*****************************************************************/
/* UDF for customizing the default Syamlal drag law in Fluent    */
/*****************************************************************/

#include "udf.h"

#define pi 4.*atan(1.)
#define diam2 3.e-4

DEFINE_EXCHANGE_PROPERTY(custom_drag, cell, mix_thread, s_col, f_col)
{
  Thread *thread_g, *thread_s;
  real x_vel_g, x_vel_s, y_vel_g, y_vel_s, abs_v, slip_x, slip_y,
          rho_g, rho_s, mu_g, reyp, afac,
          bfac, void_g, vfac, fdrgs, taup, k_g_s;

  /* find the threads for the gas (primary) */
  /* and solids (secondary phases)                    */
```

```
thread_g = THREAD_SUB_THREAD(mix_thread, s_col);/* gas phase    */
thread_s = THREAD_SUB_THREAD(mix_thread, f_col);/* solid phase*/

/* find phase velocities and properties*/

x_vel_g = C_U(cell, thread_g);
y_vel_g = C_V(cell, thread_g);

x_vel_s = C_U(cell, thread_s);
y_vel_s = C_V(cell, thread_s);

slip_x = x_vel_g - x_vel_s;
slip_y = y_vel_g - y_vel_s;

rho_g = C_R(cell, thread_g);
rho_s = C_R(cell, thread_s);

mu_g = C_MU_L(cell, thread_g);

/*compute slip*/
abs_v = sqrt(slip_x*slip_x + slip_y*slip_y);

/*compute reynolds number*/

reyp = rho_g*abs_v*diam2/mu_g;

/* compute particle relaxation time */

taup = rho_s*diam2*diam2/18./mu_g;

void_g = C_VOF(cell, thread_g);/* gas vol frac*/

/*compute drag and return drag coeff, k_g_s*/

afac = pow(void_g,4.14);

if(void_g<=0.85)
  bfac = 0.281632*pow(void_g, 1.28);
else
  bfac = pow(void_g, 9.076960);

vfac = 0.5*(afac-0.06*reyp+sqrt(0.0036*reyp*reyp+0.12*reyp*(2.*bfac-
                         afac)+afac*afac));
fdrgs = void_g*(pow((0.63*sqrt(reyp)/
```

```
                                        vfac+4.8*sqrt(vfac)/vfac),2))/24.0;

  k_g_s = (1.-void_g)*rho_s*fdrgs/taup;

  return k_g_s;

}
```

Activating an Exchange Property UDF in FLUENT

Once you have compiled and linked the source code for a custom lift or drag coefficient for the Eulerian multiphase model, you can activate it in the **Phase Interaction** panel in **FLUENT**. See Section 8.3.4 for more details.

4.4.3 DEFINE_VECTOR_EXCHANGE_PROPERTY

功能和使用方法的介绍

你可以使用 DEFINE_VECTOR_EXCHANGE_PROPERTY macro to specify custom slip velocities for the multiphase Mixture model.

Macro:	DEFINE_VECTOR_EXCHANGE_PROPERTY (name, c,
	mixture_thread, second_column_phase_index,
	first_column_phase_index, vector_result)
Argument types:	cell_t c
	Thread *mixture_thread
	int second_column_phase_index
	int first_column_phase_index
	real *vector_result
Function returns:	void

There are six arguments to DEFINE_VECTOR_EXCHANGE_PROPERTY: name, c, mixture_thread, second_column_phase_index, first_column_phase_index, and vector_result. name is the name of the UDF, specified by you. 当你的UDF编译和连接时，你为函数所选择的名字会在 FLUENT 图形用户界面中变得可见，且可被选择。 c, mixture_thread, second_column_phase_index, first_column_phase_index, and vector_result are variables that are passed by the **FLUENT** solver to your UDF.

mixture_thread is a pointer to the mixture-level thread. c is the index of a cell on the thread pointed to by mixture_thread. first_column_phase_index and second_column_phase_index are integer identifiers corresponding to the pair of phases in your multiphase flow that you are specifying a phase interaction slip velocity for. The identifiers correspond to the phases that are defined in the **Phase Interaction** panel in the graphical user interface. An index of 0 corresponds to the primary phase, and is incremented by one for each secondary phase.

Your UDF is passed the real pointer to the slip velocity vector vector_result, and it will need to set the components of the slip velocity vector (e.g., vector_result[0], vector_result[1] for a 2D problem).

例子：

下面的 UDF 名字为 custom_slip, specifies a custom slip velocity in a two-phase mixture problem.

```
/***************************************************************/
/* UDF for a defining a custom slip velocity in a 2-phase          */
/* mixture problem                                                  */
/***************************************************************/

#include "udf.h"

DEFINE_VECTOR_EXCHANGE_PROPERTY(custom_slip, c, mixture_thread,
second_column_phase_index, first_column_phase_index, vector_result)
{
  real grav[2] = {0., -9.81};
  real K = 5.e4;

  real pgrad_x, pgrad_y;

  Thread *pt, *st;/* thread pointers for primary and secondary phases*/

  pt = THREAD_SUB_THREAD(mixture_thread, second_column_phase_index);
  st = THREAD_SUB_THREAD(mixture_thread, first_column_phase_index);

  /* at this point the phase threads are known for primary (0) and
  secondary(1) phases */

  pgrad_x = C_DP(c, mixture_thread)[0];
  pgrad_y = C_DP(c, mixture_thread)[1];

  vector_result[0] =
  -(pgrad_x/K)
```

```
 +( ((C_R(c, st)-
 C_R(c, pt))/K)*
 grav[0]);

 vector_result[1] =
 -(pgrad_y/K)
 +( ((C_R(c, st)-
 C_R(c, pt))/K)*
 grav[1]);
}
```

Activating a Vector Exchange Property UDF in FLUENT

Once you have compiled and linked the source code for a custom slip velocity UDF for the Mixture multiphase model, you can activate it in the **Phase Interaction** panel in **FLUENT**. See Section 8.3.2 for more details.

4.5 DPM DEFINE Macros

This section contains descriptions of DEFINE macros for the discrete phase model (DPM). Table 4.5.1 provides a quick reference guide to the DPM DEFINE macros, the functions they define, and the panels where they are activated in **FLUENT**. Definitions of each DEFINE macro are contained in the dpm.h header file. For your convenience, these definitions are also provided in Appendix A. Note that you do not need to include the dpm.h file at the beginning of a DPM UDF, since it is already included in the udf.h file.

- DEFINE_DPM_BODY_FORCE (Section 4.5.1)
- DEFINE_DPM_DRAG (Section 4.5.2)
- DEFINE_DPM_EROSION (Section 4.5.3)
- DEFINE_DPM_INJECTION_INIT (Section 4.5.4)
- DEFINE_DPM_LAW (Section 4.5.5)
- DEFINE_DPM_OUTPUT (Section 4.5.6)
- DEFINE_DPM_PROPERTY (Section 4.5.7)
- DEFINE_DPM_SCALAR_UPDATE (Section 4.5.8)
- DEFINE_DPM_SOURCE (Section 4.5.9)
- DEFINE_DPM_SWITCH (Section 4.5.10)

Table 4.5.1: Quick Reference Guide for DPM-Specific DEFINE Macros

Variable	DEFINE **Macro**	**Panel Activated In**	
body forces on	DEFINE_DPM_BODY_FORCE	**Discrete Phase**	

particles		**Model**	
drag coefficients between	DEFINE_DPM_DRAG	**Discrete Phase Model**	
particles and fluid			
erosion and accretion rates	DEFINE_DPM_EROSION	**Discrete Phase Model**	
initializing injections	DEFINE_DPM_INJECTION_INIT	**Set Injection Properties**	
custom laws for particles	DEFINE_DPM_LAW	**Custom Laws**	
modifies what is written to the	DEFINE_DPM_OUTPUT	**Sample Trajectories**	
sampling plane output			
material properties	DEFINE_DPM_PROPERTY	**Materials**	
updates scalar every time	DEFINE_DPM_SCALAR_UPDATE	**Discrete Phase Model**	
a particle position is updated			
particle source terms	DEFINE_DPM_SOURCE	**Discrete Phase Model**	
changing the criteria for	DEFINE_DPM_SWITCH	**Custom Laws**	
switching between laws			

- 4.5.1 DEFINE_DPM_BODY_FORCE
- 4.5.2 DEFINE_DPM_DRAG
- 4.5.3 DEFINE_DPM_EROSION
- 4.5.4 DEFINE_DPM_INJECTION_INIT
- 4.5.5 DEFINE_DPM_LAW
- 4.5.6 DEFINE_DPM_OUTPUT
- 4.5.7 DEFINE_DPM_PROPERTY
- 4.5.8 DEFINE_DPM_SCALAR_UPDATE
- 4.5.9 DEFINE_DPM_SOURCE
- 4.5.10 DEFINE_DPM_SWITCH

4.5.1 DEFINE_DPM_BODY_FORCE

功能和使用方法的介绍

你可以使用 DEFINE_DPM_BODY_FORCE macro to specify a body force other than a gravitational or drag force on the particles.

Macro:	DEFINE_DPM_BODY_FORCE (name, p, i)
Argument types:	Tracked_Particle *p int i
Function returns:	real

There are three arguments to DEFINE_DPM_BODY_FORCE: name, p, and i. name is the name of the UDF, specified by you. 当你的 UDF 编译和连接时，你为函数所选择的名字会在 FLUENT 图形用户界面中变得可见，且可被选择。 p and i are variables that are passed by the **FLUENT** solver to your UDF.

p is a pointer to a Tracked_Particle structure that contains data relating to the particle being tracked. This pointer can be used as an argument to the macros defined in Section 5.7 to obtain information about particle properties (e.g., injection properties). i is an index (0, 1, or 2) that identifies the Cartesian component of the body force that is to be returned by the function. Your UDF will need to return the real value of the acceleration due to the body force (in m/s ^{2}) to the **FLUENT** solver.

例子：

下面的 UDF 名字为 particle_body_force, computes the magnetic force on a charged particle. DEFINE_DPM_BODY_FORCE is called at every particle time step in **FLUENT**, and requires a significant amount of CPU time to execute. For this reason, the UDF should be executed as a compiled UDF.

In the UDF presented below, a charged particle is introduced upstream, into a laminar flow, and travels downstream until *t*= tstart, when a magnetic field is applied. The particle takes on an approximately circular path (not an exact circular path, because the speed and magnetic force vary as the particle is slowed by the surrounding fluid).

The macro P_TIME(p) gives the current time for a particle traveling along a trajectory, which is pointed to by pointer p.

```
/*******************************************************************
***/
/* UDF for computing the magnetic force on a charged particle                    */
/*******************************************************************
***/

#include "udf.h"

#define Q 1.0                 /* particle electric charge           */
#define BZ 3.0                /* z component of magnetic field */
#define TSTART 18.0      /* field applied at t = tstart      */

/* Calculate magnetic force on charged particle.    Magnetic      */
/* force is particle charge times cross product of particle    */
/* velocity with magnetic field: Fx= q*bz*Vy,    Fy= -q*bz*Vx    */

DEFINE_DPM_BODY_FORCE(particle_body_force, p, i)
{
            real bforce;
            if(P_TIME(p)>=TSTART)
               {
                if(i==0) bforce=Q*BZ*P_VEL(p)[1];

                else if(i==1) bforce=-Q*BZ*P_VEL(p)[0];

               }
            else
                bforce=0.0;
            /* an acceleration should be returned */
            return (bforce/P_MASS(p));
}
```

Activating a DPM Body Force UDF in FLUENT

Once you have compiled and linked the source code for a custom body force UDF, you can activate it in the **Discrete Phase Model** panel in **FLUENT**. See Section 8.4.1 for more details.

4.5.2 DEFINE_DPM_DRAG

功能和使用方法的介绍

你可以使用 DEFINE_DPM_DRAG macro to specify the drag coefficient, C_D, between particles and fluid defined by the following equation:

$$F_D = \frac{18\mu}{\rho_p D_p^2} \frac{C_D \mathrm{Re}}{24}$$

Macro: DEFINE_DPM_DRAG (name, Re, p)

Argument types: real Re

Tracked_Particle *p

Function returns: real

There are three arguments to DEFINE_DPM_DRAG: name, Re, and p. name is the name of the UDF, specified by you. 当你的 UDF 编译和连接时，你为函数所选择的名字会在 FLUENT 图形用户界面中变得可见，且可被选择。 Re and p are variables that are passed by the **FLUENT** solver to your UDF. Re is the particle Reynolds number based on the particle diameter and relative gas velocity. p is a pointer to a Tracked_Particle structure that contains data relating to the particle being tracked. This pointer can be used as an argument to the macros defined in Section 5.7 to obtain information about particle properties (e.g., injection properties). i is an index (0, 1, or 2) that identifies the Cartesian component of the force that is to be returned by the function. Your UDF will need to return the real value of the drag force on a particle. The value returned to the solver will be dimensionless and represent 18 * Cd * Re / 24.

例子：

下面的 UDF 名字为 particle_drag_force, computes the drag force on a particle, and is a variation of the body force UDF presented in Section 4.5.1. The flow is the same, but a different curve is used to describe the particle drag. DEFINE_DPM_DRAG is called at every particle time step in **FLUENT**, and requires a significant amount of

CPU time to execute. For this reason, the UDF should be executed as a compiled UDF.

```
/**********************************************************************
***/
/* UDF for computing particle drag coefficient (18 Cd Re/24)                    */
/* curve as suggested by R. Clift, J. R. Grace and M.E. Weber                 */
/* "Bubbles, Drops, and Particles" (1978)                                        */
/**********************************************************************
***/

#include "udf.h"

DEFINE_DPM_DRAG(particle_drag_force, Re, p)
{
   real w, drag_force;

   if (Re < 0.01)
      {
      drag_force=18.0;
      return (drag_force);
      }
   else if (Re < 20.0)
      {
      w = log10(Re);
      drag_force = 18.0 + 2.367*pow(Re,0.82-0.05*w) ;
      return (drag_force);
      }
   else
      /* Note: suggested valid range 20 < Re < 260 */
      {
      drag_force = 18.0 + 3.483*pow(Re,0.6305) ;
      return (drag_force);
      }
}
```

Activating a DPM Drag Coefficient UDF in FLUENT

Once you have compiled and linked the source code for a custom drag coefficient UDF, you can activate it in the **Discrete Phase Model** panel in **FLUENT**. See Section 8.4.2 for more details.

4.5.3 DEFINE_DPM_EROSION

功能和使用方法的介绍

你可以使用 DEFINE_DPM_EROSION macro to access the erosion and accretion rates calculated as the particle stream strikes a wall surface. The function is called when the particle encounters a reflecting surface.

Macro: DEFINE_DPM_EROSION (name, p, t, f, normal, alpha, Vmag, mdot)

Argument types: Tracked_Particle *p

Thread *t

face_t f

real normal[]

real alpha

real Vmag

real mdot

Function returns: void

There are eight arguments to DEFINE_DPM_EROSION: name, p, t, f, normal, alpha, Vmag, and mdot. name is the name of the UDF, specified by you. 当你的 UDF 编译和连接时，你为函数所选择的名字会在 FLUENT 图形用户界面中变得可见，且可被选择。p, t, f, normal, alpha, Vmag, and mdot are variables that are passed by the **FLUENT** solver to your UDF.

p is a pointer to a Tracked_Particle structure that contains data relating to the particle being tracked. This pointer can be used as an argument to the macros defined in Section 5.7 to obtain information about particle properties (e.g., injection properties). t is a pointer to the thread that the particle is hitting. f is an index that identifies the face the particle is hitting. normal is a real array that contains the unit vector that is normal to the face. alpha is a real variable that represents the impact angle between the particle path and the face (in radians). Vmag is a real variable that represents the magnitude of the particle velocity (in m/s). mdot is the real flow rate of the particle stream as it hits the face (in kg/s).

Your UDF will need to compute the values for the erosion rate and/or accretion rate and store the values at the faces in F_STORAGE_R(f,t,SV_DPMS_EROSION) and F_STORAGE_R(f,t,SV_DPMS_ACCRETION). This is shown in the 例子： below.

例子：

The following is an 例子： of a compiled UDF that uses DEFINE_DPM_EROSION to extend postprocessing of wall impacts in a 2D axisymmetric flow. It provides additional information on how the local particle deposition rate depends on the diameter and normal velocity of the particles. It is based on the assumption that every wall impact leads to more accretion, and, therefore, every trajectory is ``evaporated'' at its first wall impact. (This is done by first setting a DPM user scalar within DEFINE_DPM_EROSION, which is then evaluated in a DEFINE_DPM_LAW, which sets P_MASS to zero.) User-defined memory locations (UDMLs) are used to store and visualize the following:

- number of wall impacts since UDMLs were reset. (Resetting is typically done at the beginning of a **FLUENT** session by the use of DEFINE_ON_DEMAND in order to avoid the use of uninitialized data fields. Resetting prevents the addition of sampled data being read from a file.)
- average diameter of particles hitting the wall.
- average radial velocity of particles.

Before tracing the particles, you will have to reset the UDMLs and assign the global domain pointer by executing the DEFINE_ON_DEMAND function.

```
/*********************************************************************
***/
/*    UDF for extending postprocessing of wall impacts                              */
/*********************************************************************
***/
#include "udf.h"

#define MIN_IMPACT_VELO -1000.
   /* Minimum particle velocity normal to wall (m/s) to allow Accretion.*/

Domain *domain; /* Get the domain pointer and assign it later to domain*/

enum                        /* Enumeration of used User Defined Memory Locations. */
{
   NUM_OF_HITS,        /* Number of particle hits into wall face considered.*/
   AVG_DIAMETER,        /* Average diameter of particles that hit the wall. */
   AVG_RADI_VELO,        /* Average radial velocity of ""    ""    ------------ */
   NUM_OF_USED_UDM
};
```

```
int UDM_checked = 0;          /* Availability of UDMLs checked? */

void reset_UDM_s(void);    /* Function to follow below. */

int
check_for_UDM(void)           /* Check for UDMLs' availability... */
{
   Thread *t;

   if (UDM_checked)
      return UDM_checked;

   if (!rp_axi)
     Internal_Error("UDF-Error: only valid for 2d-axisymmetric cases!\n");

   thread_loop_c(t, domain)       /* We require all cell threads to.. */
      {                                      /* provide space in memory for UDML */
         if (FLUID_THREAD_P(t))
            if (NULLP(THREAD_STORAGE(t,SV_UDM_I)))
                  return 0;
      }

   UDM_checked = 1;          /* To make the following work properly... */
   reset_UDM_s();            /* This line will only be executed once, */
   return UDM_checked;    /* because check_for_UDM checks... */
}                                    /* ...for UDM_checked first. */

void
reset_UDM_s(void)
{
   Thread *t;
   cell_t    c;
   face_t    f;
   int         i;

   if (!check_for_UDM())    /* Don't do it, if memory is not available. */
      return;

   CX_Message("Resetting User Defined Memory...\n");

   thread_loop_f(t, domain)
      {
         if (NNULLP(THREAD_STORAGE(t,SV_UDM_I)))
```

```
            {
              begin_f_loop(f, t)
                {
                  for (i = 0; i < NUM_OF_USED_UDM; i++)
                    F_UDMI(f, t, i) = 0.;
                }
              end_f_loop(f, t)
            }
          else
            {
              CX_Message("    Skipping FACE thread no. %d..\n", THREAD_ID(t));
            }
        }
  thread_loop_c(t, domain)
        {
          if (NNULLP(THREAD_STORAGE(t,SV_UDM_I)))
            {
              begin_c_loop(c, t)
                {
                  for (i = 0; i < NUM_OF_USED_UDM; i++)
                    C_UDMI(c, t, i) = 0.;
                }
              end_c_loop(c, t)
            }
          else
            {
              CX_Message(" Skipping CELL thread no. %d..\n", THREAD_ID(t));
            }
        }                   /* Skipping Cell Threads can happen if the user */
                                  /* uses reset_UDM prior to initializing. */
  CX_Message(" --- Done.\n");
}

DEFINE_DPM_SCALAR_UPDATE(dpm_scalup, c, t, if_init, p)
{   if (if_init)
      P_USER_REAL(p, 0) = 0;      /* Simple initialization. Used later for
                                   * stopping trajectory calculation */
}

DEFINE_DPM_EROSION(dpm_accr, p, t, f, normal, alpha, Vmag, Mdot)
{
  real A[ND_ND], area;
  int num_in_data;
  Thread *t0;
```

```
  cell_t    c0;

  real radi_pos[2], radius, imp_vel[2], vel_ortho;

/* The following is ONLY valid for 2d-axisymmetric calculations!!! */
/* Additional effort is necessary because DPM tracking is done in    */
/* THREE dimensions for TWO-dimensional axisymmetric calculations. */

  radi_pos[0] = p->state.pos[1];           /* Radial location vector. */
  radi_pos[1] = p->state.pos[2];           /* (Y and Z in 0 and 1...) */

  radius = NV_MAG(radi_pos);
  NV_VS(radi_pos, =, radi_pos, /, radius);
                                            /* Normalized radius direction vector.*/
  imp_vel[0] = P_VEL(p)[0];        /* Axial particle velocity component. */
  imp_vel[1] = NVD_DOT(radi_pos, P_VEL(p)[1], P_VEL(p)[2], 0.);
  /* Dot product of normalized radius vector and y & z components */
  /* of particle velocity vector gives _radial_ particle velocity */
  /* component    */
  vel_ortho = NV_DOT(imp_vel, normal); /*velocity orthogonal to wall */

  if (vel_ortho < MIN_IMPACT_VELO)    /* See above, MIN_IMPACT_VELO */
    return;

  if (!UDM_checked)             /* We will need some UDM's, */
    if (!check_for_UDM()) /* so check for their availability.. */
      return;                         /* (Using int variable for speed, could */
                                    /*    even just call check_for UDFM().) */
  c0 = F_C0(f, t);
  t0 = F_C0_THREAD(f, t);

  num_in_data = F_UDMI(f, t, NUM_OF_HITS);

   /* Add particle to statistics: Calculate...:

   current_particle_property +
   earlier_particles_averaged_property * number_of_earlier_particles
   ---------------------------------------------------------------
                         number_of_earlier_particles + 1                              */

/* Average diameter of particles that hit the particular wall face:*/
  F_UDMI(f, t, AVG_DIAMETER) = (P_DIAM(p)
                                   +        num_in_data     *     F_UDMI(f,     t,
AVG_DIAMETER))
```

```
                                        / (num_in_data + 1);
  C_UDMI(c0,t0,AVG_DIAMETER) = F_UDMI(f, t, AVG_DIAMETER);

 /* Average velocity normal to wall of particles hitting the wall:*/
  F_UDMI(f, t, AVG_RADI_VELO) = (vel_ortho
                                  +   num_in_data * F_UDMI(f, t, AVG_RADI_VELO))
                                      / (num_in_data + 1);
  C_UDMI(c0,t0,AVG_RADI_VELO) = F_UDMI(f, t, AVG_RADI_VELO);

  F_UDMI(f, t, NUM_OF_HITS) = num_in_data + 1;
  C_UDMI(c0,t0,NUM_OF_HITS) = num_in_data + 1;

  F_AREA(A, f, t);
  area = NV_MAG(A);
  F_STORAGE_R(f,t,SV_DPMS_ACCRETION) += Mdot / area;
                                                    /* copied from source. */

  P_USER_REAL(p, 0) = 1.;         /* "Evaporate" */
}

DEFINE_DPM_LAW(stop_dpm_law, p, if_cpld)
{
  if (0. < P_USER_REAL(p, 0))
     P_MASS(p) = 0.;                 /* "Evaporate" */
}

DEFINE_ON_DEMAND(reset_UDM)
{
  /* assign domain pointer with global domain */
  domain = Get_Domain(1);
  reset_UDM_s();
}
```

Activating an Erosion/Accretion UDF in FLUENT

Once you have compiled and linked the source code for a custom erosion/accretion UDF, you can activate it in the **Discrete Phase Model** panel in **FLUENT**. See Section 8.4.3 for more details.

4.5.4 DEFINE_DPM_INJECTION_INIT

功能和使用方法的介绍

你可以使用 DEFINE_DPM_INJECTION_INIT macro when you want to initialize injections by specifying the particle properties at the time of injection.

Macro: DEFINE_DPM_INJECTION_INIT (name, I)

Argument types: Injection *I

Function returns: void

There are two arguments to DEFINE_DPM_INJECTION_INIT: name and I. name is the name of the UDF, specified by you. 当你的 UDF 编译和连接时，你为函数所选择的名字会在 FLUENT 图形用户界面中变得可见，且可被选择。 I is a variable that is passed by the **FLUENT** solver to your UDF.

I is a pointer to the Injection structure which is a container for the particles being created. This function is called twice for each Injection before the first DPM iteration, and then called once for each Injection before the particles are injected into the domain at each subsequent DPM iteration.

Your UDF can initialize a particle's injection properties such as location, diameter, and velocity.

例子：

下面编译的 UDF 名字为 init_random_diameter, initializes the particle diameter (P_DIAM) with a randomized distribution in the range from 10^{-5} m to 10^{-3} m. The initial particle density (P_RHO) is set to 1 kg/m^3. Initial particle mass is computed accordingly.

```
/******************************************************************
*/
/* UDF that initializes particle injection properties                    */
/******************************************************************
*/

#include "udf.h"
#include "random.h"      /* not included in udf.h so must include here */


DEFINE_DPM_INJECTION_INIT(init_random_diameter,I)
```

```
{
  Particle *p;

  CX_Message("Initializing Injection: %s with random diameter\n",I->name);

  loop(p,I->p)    /* Standard Fluent Looping Macro to get
particle streams in an Injection */
  {
     p->flow_rate = 1.e-4;
     P_DIAM(p) = 1.e-5 + fabs(cheap_gauss_random())*1.e-3;
     P_RHO(p) = 1.0;
     P_MASS(p) = P_RHO(p)*M_PI*pow(P_DIAM(p),3.0)/6.0;
  }
}
```

Activating a DPM Initialization UDF in FLUENT

Once you have compiled and linked the source code for a DPM initialization UDF, you can activate it in the **Set Injection Properties** panel in **FLUENT**. See Section 8.4.4 for more details.

4.5.5 DEFINE_DPM_LAW

功能和使用方法的介绍

你可以使用 DEFINE_DPM_LAW macro when you want to customize laws for particles. The function can specify the heat and mass transfer rates for droplets and combusting particles.

Macro:	DEFINE_DPM_LAW (name, p, ci)
Argument types:	Tracked_Particle *p
	int ci
Function returns:	void

There are three arguments to DEFINE_DPM_LAW: name, p, and ci. name is the name of the UDF, specified by you. 当你的 UDF 编译和连接时，你为函数所选择的名字会在 FLUENT 图形用户界面中变得可见，且可被选择。 p and ci are variables that are passed by the **FLUENT** solver to your UDF.

p is a pointer to a Tracked_Particle structure that contains data relating to the particle being tracked. This pointer can be used as an argument to the macros defined in Section 5.7 to obtain information about particle properties (e.g., injection properties). ci is an integer that indicates whether the continuous and discrete phases are coupled (equal to 1 if coupled with continuous phase, 0 if not coupled).

Your UDF can define custom laws for particle properties (mass, diameter, temperature, etc.) as the droplet or particle exchanges mass and energy with its surroundings.

例子：

下面的 UDF 名字为 Evapor_Swelling_Law, models a custom law for the evaporation swelling of particles. It can be executed as an interpreted or compiled UDF in **FLUENT**. (See Section 4.5.10 for another 例子： of DEFINE_DPM_LAW usage.)

```
/*******************************************************************
**/
/* UDF that models a custom law for evaporation swelling of particles */
/*******************************************************************
**/

#include "udf.h"

DEFINE_DPM_LAW(Evapor_Swelling_Law, p, ci)
{
   real swelling_coeff = 1.1;

   /* first, call standard evaporation routine to calculate *
     * the mass and heat transfer                          */
   VaporizationLaw(p);
   /* compute new particle diameter and density */
   P_DIAM(p) = P_INIT_DIAM(p)*(1. + (swelling_coeff - 1.)*

(P_INIT_MASS(p)-P_MASS(p))/(DPM_VOLATILE_FRACTION(p)*P_INIT_MAS
S(p)));
   P_RHO(p) = P_MASS(p) / (3.14159*P_DIAM(p)*P_DIAM(p)*P_DIAM(p)/6);
   P_RHO(p) = MAX(0.1, MIN(1e5, P_RHO(p)));
}
```

Activating a Custom DPM Law in FLUENT

Once you have compiled and linked the source code for a custom DPM law UDF, you can activate it in the **Custom Laws** panel, which is opened from the **Set Injection Properties** panel. See Section 8.4.5 for more details.

4.5.6 DEFINE_DPM_OUTPUT

功能和使用方法的介绍

你可以使用 DEFINE_DPM_OUTPUT macro when you want to modify what is written out to the sampling plane output. This function allows access to the variables that are written as a particle passes through a sample plane (see Chapter 19 of the User's Guide).

Macro:	DEFINE_DPM_OUTPUT (name, header, fp, p, t, plane)
Argument types:	int header
	FILE *fp
	Tracked_Particle *p
	Thread *t
	Plane *plane
Function returns:	void

There are six arguments to DEFINE_DPM_OUTPUT: name, header, fp, p, t, and plane. name is the name of the UDF, specified by you. 当你的 UDF 编译和连接时，你为函数所选择的名字会在 FLUENT 图形用户界面中变得可见，且可被选择。header, fp, p, t, and plane are variables that are passed by the **FLUENT** solver to your UDF.

header is an integer that is equal to 1 at the first call of the function before particles are tracked and set to 0 for subsequent calls. fp is a pointer to the file to or from which you are writing or reading. p is a pointer to a Tracked_Particle structure that contains data relating to the particle being tracked. This pointer can be used as an argument to the macros defined in Section 5.7 to obtain information about particle properties (e.g., injection properties).

t is a pointer to the thread that the particle is passing through if it is a grid surface. If the particle is not passing through a grid surface, then the value of t is NULL. plane is a pointer to the Plane structure (see dpm.h) if the particle is passing through a planar slice (line in 2D). If the particle is passing through a grid surface, then plane is NULL. The output of your UDF will be written to the file indicated by fp.

例子：

See Section 4.5.8 for an 例子： of a UDF that uses the DEFINE_DPM_OUTPUT macro.

Activating a DPM Output UDF in FLUENT

Once you have compiled and linked the source code for a DPM output UDF, you can activate it in **FLUENT** in the **Sample Trajectories** panel. See Section 8.4.6 for more details.

4.5.7 DEFINE_DPM_PROPERTY

功能和使用方法的介绍

你可以使用 DEFINE_DPM_PROPERTY macro when you want to specify properties of discrete phase materials.

Macro:	DEFINE_DPM_PROPERTY (name, c, t, p)
Argument types:	cell_t c
	Thread *t
	Tracked_Particle *p
Function returns:	real

There are four arguments to DEFINE_DPM_PROPERTY: name, c, t, and p. DEFINE_DPM_PROPERTY has the same arguments as the DEFINE_PROPERTY function (described in Section 4.3.6), with the addition of the pointer to the Tracked_Particle p.

name is the name of the UDF, specified by you. 当你的 UDF 编译和连接时，你为函数所选择的名字会在 FLUENT 图形用户界面中变得可见，且可被选择。 c, t, and p are variables that are passed by the **FLUENT** solver to your UDF.

c is an index that identifies the cell where the particle is located in the given thread. t is a pointer to the thread where the particle is located. p is a pointer to a Tracked_Particle structure that contains data relating to the particle being tracked. This pointer can be used as an argument to the macros defined in Section 5.7 to obtain information about particle properties (e.g., injection properties). Your UDF will need to compute the real value of the discrete phase property and return it to the **FLUENT** solver.

You can model the following dispersed phase properties with this type of UDF:

- particle emissivity
- vapor pressure
- vaporization temperature
- particle scattering factor
- boiling point
- particle viscosity
- particle surface tension

例子：

在下面的例子中，：，two discrete phase material property UDFs (named coal_emissivity and coal_scattering, respectively) are concatenated into a single C source file. These UDFs can be executed as compiled UDFs in **FLUENT**.

```
/*******************************************************************
*/
/* UDF that specifies discrete phase materials                    */
/*******************************************************************
*/

#include "udf.h"

DEFINE_DPM_PROPERTY(coal_emissivity, c, t, p)
{
  real mp0= P_INIT_MASS(p);
  real mp = P_MASS(p);
  real vf, cf;

  /* get the material char and volatile fractions and store them */
  /* in vf and cf                                                 */
  vf=DPM_VOLATILE_FRACTION(p);
  cf=DPM_CHAR_FRACTION(p);

  if (!(((mp/mp0) >= 1) || ((mp/mp0) <= 0)))
    {
      if ((mp/mp0) < (1-(vf)-(cf)))
        {
          /* only ash left */
          /* vf = cf = 0; */
          return .001;
        }
      else if ((mp/mp0) < (1-(vf)))
        {
          /* only ash and char left */
          /* cf = 1 - (1-(vf)-(cf))/(mp/mp0); */
```

```
            /* vf = 0; */
            return 1.0;
          }
        else
          {
            /* volatiles, char, and ash left */
            /* cf =    (cf)/(mp/mp0); */
            /* vf = 1. - (1.-(vf))/(mp/mp0); */
            return 1.0;
          }
      }
  return 1.0;
}

DEFINE_DPM_PROPERTY(coal_scattering, c, t, p)
{
  real mp0= P_INIT_MASS(p);
  real mp = P_MASS(p);
  real cf, vf;

  /* get the original char and volatile fractions and store them */
  /* in vf and cf                                                  */
  vf=DPM_VOLATILE_FRACTION(p);
  cf=DPM_CHAR_FRACTION(p);

  if (!(((mp/mp0) >= 1) || ((mp/mp0) <= 0)))
    {
      if ((mp/mp0) < (1-(vf)-(cf)))
        {
          /* only ash left */
          /* vf = cf = 0; */
          return 1.1;
        }
      else if ((mp/mp0) < (1-(vf)))
        {
          /* only ash and char left */
          /* cf = 1 - (1-(vf)-(cf))/(mp/mp0); */
          /* vf = 0; */
          return 0.9;
        }
      else
        {
          /* volatiles, char, and ash left */
          /* cf =    (cf)/(mp/mp0); */
```

```
            /* vf = 1. - (1.-(vf))/(mp/mp0); */
            return 1.0;
         }
      }
   return 1.0;
}
```

Activating a DPM Material Property UDF in FLUENT

Once you have compiled and linked your property UDF, you can activate it by selecting it in the **Materials** panel in **FLUENT**. See Section 8.4.7 for more details.

4.5.8 DEFINE_DPM_SCALAR_UPDATE

功能和使用方法的介绍

你可以使用 DEFINE_DPM_SCALAR_UPDATE macro to update a scalar quantity every time a particle position is updated. The function allows particle-related variables to be updated or integrated over the life of the particle. Particle values can be stored in an array associated with the Tracked_Particle (accessed with the macro P_USER_REAL(p,i)). Values calculated and stored in the array can be used to color the particle trajectory.

Macro:	DEFINE_DPM_SCALAR_UPDATE (name, c, t, initialize, p)
Argument types:	cell_t c
	Thread *t
	int initialize
	Tracked_Particle *p
Function returns:	void

There are five arguments to DEFINE_DPM_SCALAR_UPDATE: name, c, t, initialize, and p. name is the name of the UDF, specified by you. 当你的 UDF 编译和连接时，你为函数所选择的名字会在 FLUENT 图形用户界面中变得可见，且可

被选择。 c, t, initialize, and p are variables that are passed by the **FLUENT** solver to your UDF.

c is an index that identifies the cell that the particle is currently in. t is a pointer to the thread the particle is currently in. initialize is an integer that has a value of 1 when the function is called at the start of the particle integration, and 0 thereafter. p is a pointer to a Tracked_Particle structure that contains data relating to the particle being tracked. This pointer can be used as an argument to the macros defined in Section 5.7 to obtain information about particle properties (e.g., injection properties).

The DEFINE_DPM_SCALAR_UPDATE function is called at the start of particle integration (when initialize is equal to 1) and then after each time step for the particle trajectory integration. The real array user is available for storage. The size of the array should be set in the **Discrete Phase Model** panel in the **Number of Scalars** field.

例子：

The following is an 例子：of a compiled UDF that computes the melting index along a particle trajectory. The DEFINE_DPM_SCALAR_UPDATE function is called at every particle time step in **FLUENT**, and requires a significant amount of CPU time to execute.

The melting index is computed from

$$\text{melting index} = \int_0^t \frac{1}{\mu} dt \tag{4.5.1}$$

Also included in this UDF is an initialization function DEFINE_INIT that is used to initialize the scalar variables. The DPM_OUTPUT function is used to write out the melting index at sample planes and surfaces. The macro NULL_P, which expands to ((p) == NULL), checks if its argument is a null pointer.

```
/*********************************************************************
*/
/* UDF for computing the melting index along a particle trajectory      */
/*********************************************************************
*/

#include "udf.h"
```

```
static real viscosity_0;

DEFINE_INIT(melt_setup, domain)
{
        /* if memory for the particle variable titles has not been
         * allocated yet, do it now */

  if (NULLP(user_particle_vars)) Init_User_Particle_Vars();

         /* now set the name and label */

  strcpy(user_particle_vars[0].name,"melting-index");
  strcpy(user_particle_vars[0].label,"Melting Index");
}



        /* update the user scalar variables */

DEFINE_DPM_SCALAR_UPDATE(melting_index, cell, thread, initialize, p)
{
  cphase_state_t *c = &(p->cphase);
  if (initialize)
    {
     /* this is the initialization call, set:
      * p->user[0] contains the melting index, initialize to 0
      * viscosity_0 contains the viscosity at the start of a time step*/

      p->user[0] = 0.;
      viscosity_0 = c->mu;
    }

  else
    {
      /* use a trapezoidal rule to integrate the melting index */
      p->user[0] += P_DT(p) * .5 * (1/viscosity_0 + 1/c->mu);

      /* save current fluid viscosity for start of next step */
      viscosity_0 = c->mu;
    }
}

    /* write melting index when sorting particles at surfaces */
DEFINE_DPM_OUTPUT(melting_output, header, fp, p, thread, plane)
{
```

```
   char name[100];

   if (header)
     {
     if (NNULLP(thread))
         cxprintf(fp,"(%s %d)\n",thread->head->dpm_summary.sort_file_name,11);
     else
         cxprintf(fp,"(%s %d)\n",plane->sort_file_name,11);
     cxprintf(fp,"(%10s %10s    %10s    %10s    %10s    %10s    %10s"
                        " %10s    %10s    %10s    %10s    %s)\n",
                           "X","Y","Z","U","V","W","diameter","T","mass-flow",
                           "time","melt-index","name");
       }
   else
       {
           sprintf(name,"%s:%d",p->injection->name,p->part_id);
           cxprintf(fp,
                        "((%10.6g    %10.6g    %10.6g    %10.6g    %10.6g    %10.6g    "
                        "%10.6g    %10.6g    %10.6g    %10.6g %10.6g) %s)\n",
                        p->state.pos[0], p->state.pos[1], p->state.pos[2],
                        p->state.V[0], p->state.V[1], p->state.V[2],
                        p->state.diam, p->state.temp, p->flow_rate, p->state.time,
                        p->user[0], name);
       }
}
```

Activating a DPM Scalar Update UDF in FLUENT

Once you have compiled and linked your property UDF, you can activate it by selecting it in the **Discrete Phase Model** panel in **FLUENT**. See Section 8.4.8 for more details.

4.5.9 DEFINE_DPM_SOURCE

功能和使用方法的介绍

你可以使用 DEFINE_DPM_SOURCE macro to access particle source terms. The function allows access to the accumulated source terms for a particle in a given cell before they are added to the mass, momentum, and energy exchange terms for coupled DPM calculations.

Macro:	DEFINE_DPM_SOURCE (name, c, t, S, strength, p)

Argument types:	cell_t c
	Thread *t
	dpms_t *S
	real strength
	Tracked_Particle *p
Function returns:	void

There are six arguments to DEFINE_DPM_SOURCE: name, c, t, S, strength, and p. name is the name of the UDF, specified by you. 当你的 UDF 编译和连接时，你为函数所选择的名字会在 FLUENT 图形用户界面中变得可见，且可被选择。c, t, S, strength, and p are variables that are passed by the **FLUENT** solver to your UDF.

c is an index that identifies the cell the particle is currently in. t is a pointer to the thread the particle is currently in. S is a pointer to the source structure dpms_t, which contains the source terms for the cell. strength is the particle number flow rate in particles/second (divided by the number of tries if stochastic tracking is used). p is a pointer to a Tracked_Particle structure that contains data relating to the particle being tracked. This pointer can be used as an argument to the macros defined in Section 5.7 to obtain information about particle properties (e.g., injection properties). The modified source terms, once computed by the function, will be stored in S.

例子：

See Section 4.5.10 for an 例子： of DEFINE_DPM_SOURCE usage.

Activating a DPM Source Term UDF in FLUENT

Once you have compiled and linked your source term UDF, you can activate it by selecting it in the **Fluid** panel in **FLUENT**. See Section 8.4.9 for more details.

4.5.10 DEFINE_DPM_SWITCH

功能和使用方法的介绍

你可以使用 DEFINE_DPM_SWITCH macro to modify the criteria for switching between laws. The function can be used to control the switching between the user-defined particle laws and the default particle laws, or between different user-defined or default particle laws.

Macro:	DEFINE_DPM_SWITCH (name, p, ci)
Argument types:	Tracked_Particle *p
	int ci
Function returns:	void

There are three arguments to DEFINE_DPM_SWITCH: name, p, and ci. name is the name of the UDF, specified by you. 当你的 UDF 编译和连接时，你为函数所选择的名字会在 FLUENT 图形用户界面中变得可见，且可被选择。 p and ci are variables that are passed by the **FLUENT** solver to your UDF.

p is a pointer to a Tracked_Particle structure that contains data relating to the particle being tracked. This pointer can be used as an argument to the macros defined in Section 5.7 to obtain information about particle properties (e.g., injection properties). ci is an integer that indicates if the continuous and discrete phases are coupled (equal to 1 if coupled with continuous phase, 0 if not coupled).

例子：

The following is an 例子：of a compiled UDF that uses DEFINE_DPM_SWITCH to switch between DPM laws using a criterion. The UDF switches to DPM_LAW_USER_1 which refers to condenshumidlaw since only one user law has been defined. The switching criterion is the local humidity which is computed in the domain using a DEFINE_ON_DEMAND function, which again calls the function myHumidity for every cell. In the case where the humidity is greater than 1, condensation is computed by applying a simple mass transfer calculation. Otherwise, one of **FLUENT**'s standard laws for Vaporization or Inert Heating are applied, depending on the particle mass. The UDF requires one UDML, and needs a species called h2o to compute the local humidity.

```
/*********************************************************************
**/
/* Concatenated UDFs for the Discrete Phase Model that includes a        */
/*                usage                of                DPM_SWITCH
*/
/*********************************************************************
**/

#include "udf.h"
#include "surf.h"    /* not included in udf.h so must include here */

/* include surf.h for macros:
  * RP_Cell() & RP_Thread()
```

```
  */

real H2O_Saturation_Pressure(real T)
{
   real ratio, aTmTp;

   aTmTp = .01 * (T - 338.15);
   ratio = (647.286/T - 1.) *
      (-7.419242 + aTmTp*(.29721 +
                  aTmTp*(-.1155286 +
                  aTmTp*(8.685635e-3 +
                  aTmTp*(1.094098e-3 +
                  aTmTp*(-4.39993e-3 +
                  aTmTp*(2.520658e-3 -
                  aTmTp*5.218684e-4)))))));
   return (22.089e6 * exp(MIN(ratio,35.)));
}

real myHumidity(cell_t c, Thread *t)
{
   int i;
   Material *m=THREAD_MATERIAL(t,0), *sp;
   real yi_h2o,mw_h2o;
   real r_mix=0.0;

   if(MATERIAL_TYPE(m)==MATERIAL_MIXTURE)
      {
         mixture_species_loop (m,sp,i)
            {
               r_mix += C_YI(c,t,i)/MATERIAL_PROP(sp,PROP_mwi);

               if (0 == strcmp(MIXTURE_SPECIE_NAME(m,i),"h2o") ||
                     (0 == strcmp(MIXTURE_SPECIE_NAME(m,i),"H2O")))
                  {
                     yi_h2o = C_YI(c,t,i);
                     mw_h2o = MATERIAL_PROP(sp,PROP_mwi);
                  }
            }
      }

   return ((ABS_P(C_P(c,t),op_pres) * yi_h2o / (mw_h2o * r_mix)) /
                     H2O_Saturation_Pressure(C_T(c,t))) ;
}
```

```
#define CONDENS 2.0
#define MINMASS 1.0e-9

DEFINE_DPM_LAW(condenshumidlaw,p,coupled)
{
  real area = M_PI * (P_DIAM(p)*P_DIAM(p)); /*M_PI is accurate PI Value*/
  real mp_dot;
  cell_t c = RP_CELL(&p->cCell);          /* Get Cell and Thread from */
  Thread *t = RP_THREAD(&p->cCell);    /* Particle Structure */

  /* This law only used if Humidity > 1.0) */
  mp_dot = CONDENS*area*(myHumidity(c,t)-1.0);

  if(mp_dot>0.0)
  {
     P_MASS(p) = MAX((P_MASS(p) + mp_dot*(p->time_step)),MINMASS);
     P_DIAM(p) = pow(P_MASS(p) * 6. / (P_RHO(p) * M_PI), 1./3.);
     P_T(p)=C_T(c,t); /* Assume condensing particle is in thermal
                                    equilibrium with fluid in cell */
  }
}

DEFINE_DPM_SOURCE(dpm_source, c, t, S, strength, p)
{
   real mp_dot;
   Material *sp = p->injection->material;

   if (P_CURRENT_LAW(p) == DPM_LAW_USER_1)
      {
        /* mp_dot is the (positive) mass source to the continuous phase */
        /* (Difference in mass between entry and exit from cell)                 */
        /* multiplied by strength (Number of particles/s in stream)           */

        mp_dot = (P_MASS0(p) - P_MASS(p)) * strength;

        S->mass +=    mp_dot;

        /* add latent heat of vaporization to dpm energy source    */
        S->energy -=    mp_dot * MATERIAL_PROP(sp,PROP_latent_heat);

        /* Sensible heat dealt with by Fluent Automatically */
      }
}
```

```
DEFINE_DPM_SWITCH(dpm_switch,p,coupled)
{
  cell_t c = RP_CELL(&p->cCell);
  Thread *t = RP_THREAD(&p->cCell);

  if(myHumidity(c,t) > 1.0)
    P_CURRENT_LAW(p) = DPM_LAW_USER_1;
  else
  {
    if(P_MASS(p) <= MINMASS)
      P_CURRENT_LAW(p) = DPM_LAW_INITIAL_INERT_HEATING;
    else
      P_CURRENT_LAW(p) = DPM_LAW_VAPORIZATION;
  }

}

#define UDM_RH 0
#define N_REQ_UDM 1

DEFINE_ON_DEMAND(set_relhum)
{
  Domain *domain = Get_Domain(1);
  cell_t cell;
  Thread *thread;

  if(sg_udm<N_REQ_UDM)
    CX_Message("\nNot enough user defined memory allocated. %d required\n",
                    N_REQ_UDM);
  else
  {
    thread_loop_c(thread,domain)
    {
      /* Check if thread is a Fluid thread and has UDMs set up on it */
      if                                              (FLUID_THREAD_P(thread)&&
NNULLP(THREAD_STORAGE(thread,SV_UDM_I)))
      {
        begin_c_loop(cell,thread)
          C_UDMI(cell,thread,UDM_RH)=myHumidity(cell,thread);
        end_c_loop(cell,thread)
      }
    }
    CX_Message("\nRelative Humidity set in udm-%d\n",UDM_RH);
  }/* end if for enough UDSs and UDMs */
```

```
}
```

Activating a DPM Switching UDF in FLUENT

Once you have compiled and linked the source code for a DPM switching UDF, you can activate it in **FLUENT** in the **Custom Laws** panel, which is opened from the **Set Injection Properties** panel. See Section 8.4.5 for more details.

第五章 使用宏存取FLUENT解算器变量

本章提供了可以存取FLUENT解算器中变量的预定义宏。

5.1 介绍
5.2 单元宏
5.3 面宏
5.4 几何图形宏
5.5 节点宏
5.6 多相宏
5.7 DPM宏
5.8 NO_x宏

5.1 Introduction

Fluent Inc. has provided you with a set of predefined functions that you can use to access data from the FLUENT solver. These functions are primarily implemented in the code as macros. The macros listed in this chapter are defined in header files such asmem.h, metric.h, and dpm.h. The udf.h file contains definitions for DEFINE macros,as well as #include directives for most of the solver access macro header files found inthis chapter. Therefore, including udf.h in your source code file will also result in theinclusion of solver access .h files.Some examples of solver data you can access using predefined macros are:

_ solution variables and their derivatives (e.g., velocity, temperature, turbulencequantities)
_ geometry variables (e.g., coordinates, areas, volumes)
_ grid and node variables (e.g., node velocities)
_ material property variables (e.g., density, viscosity, conductivity)
_ discrete phase model variables

5.1 介绍

Fluent公司提供了一系列预定义函数来从求解器中读写数据。这些函数以宏的形式存放在代码中。这章的所列出的宏是被定义在扩展名为.h文件里的。例如mem.h, metric.h, 和dpm.h。在udf.h文件中包含了宏的定义和这章中所用到的大部分宏文件和它的说明。因此如果在你的原程序中包含了udf.h文件，那么也就包含了各种的求解器读写文件了。（.h文件）。

下面列出了一些使用预先设计的宏来读写数据类型：

——溶液变量及它们的组合变量（速度，温度，湍流量等）

——几何变量（坐标，面积，体积等）

——网格和节点变量（节点速度等）

——材料性质变量（密度，粘度，导电性等）

——分散相模拟变量。
For all types of data except specific heat, the word \access" refers to reading and writing data. In the case of specific heat, however, data can be read but cannot be modified.
In the following sections, each macro is listed with its arguments, argument types, and returned value. Arguments belong to the following data types:
cell_t c cell identifier
face_t f face identifier
Thread *t pointer to a thread
Thread **pt pointer to an array of phase threads
Int i integer index
Node *node pointer to a node
Arguments are either inputs to a function, or are outputs. Each macro returns a value, which either is output back to the solver (as an argument), or is available for assignment in your function.

对于除了指定的热量以外的所有数据而言，存取这个词还指读写数据。对于指定的的热量的数据是只能读不能改的。在下面章节中列出了每一个宏的包含的参数，参数的类型和返回值。其中参数属于下面的数据类型。

cell_t c 单元格标识符

face_t 面积标识符

Thread *t 线指示器

Thread **pt 象限矩阵指示器

Int I 整数
Node *node 节点指示器
参数既不是方程的输入也不是方程的输出。每一个宏返回一个值，这个值要么作为一个参数以输出值的形式返回求解器，要么是你方程中可用参数。

For example, the macro C_T
real temp;
temp = C_T(c,t);
has two arguments, cell identifier c and thread pointer t. These arguments are passed from the FLUENT solver to the function. C T returns the real value of temperature, which can then be assigned to a variable in your UDF (temp in this example).

例如宏C_T
real temp;
temp = C_T(c,t);

C_T有两个参数，单元标识符C和线指示器t。这些参数从FLUENT求解器中返回到方程中。C_T返回一个温度的实数值，这个值能够分配到的你的UDF变量中去。（比如这个例子中的temp）

C_CENTROID(x,c,t);
has three arguments: x, c, and t. In this case the cell identifier c and thread pointer t are input arguments, while the array x (the cell centroid) is output to the FLUENT solver as an argument.
C_CENTROID(x,c,t)有三个参数，x,c,t。在这里单元标识符C和线指示器是输入参数，而矩阵x（单元格的质心）是以参数形式输出到FLUENT求解器中的。

5.2 Cell Macros

The macros listed in this section are real variables returned by the solver, and are de_ned on a cell. These cell variables are available in both the segregated and the coupled solvers. Definitions for these cell macros can be found in the referenced header files (e.g., mem.h).

5.2 单元格宏
这章所列出的宏是由求解器返回的实数变量，并且这些变量都是定义在一个单元格中的。这些单元格变量在单独的或是联合的求解器中都能得到。(segregated and the coupled solvers.）
这些单元格宏的定义在相关的扩展名为（.h）的文件中可以得到。（例如mem.h等）

5.2.1 Macros for Accessing Flow Variables

Macros for accessing flow variables in FLUENT are shown in Table 5.2.1. Note that the G, RG, M1, and M2 suffixes that are shown for the cell temperature macro (C T) can be applied to all of the solver variable macros listed in Table 5.2.1 with the exception of cell pressure (C P). These suffixes correspond to gradient vector, reconstruction gradient vector, previous time step, and 2nd previous time step, respectively. In the case of cell pressure, the gradient vector and components are derived using C DP instead of C P G. A description of each suffix and its usage is presented in the sections below.

5.2.1 用来读写流体变量的宏。

在FLUENT中可以用来读写流体变量的宏在表5.2.1中列出，注意加了_G,_ RG, _M1,和_M2这些下标的单元格温度的宏可以应用于表5.2.1 中的所有求解器的变量中，但是除了单元格压力（C-P）。这些下标分别表示的是矢量梯度，改造的矢量梯度，前一次的步长，和前两次的步长。而对于单元格压力，它的矢量梯度和相应的分量是使用C_DP 得到的而不是C_P_G。每一个下标的描述和用法在下面介绍。

表 5.2.1: 在mem.h文件中的流体变量宏

名称（参数）	参数类型	返回值
C_T(c,t)	cell t c, Thread *t	温度
C_T_G(c,t)	cell t c, Thread *t	温度梯度矢量
C_T_G(c,t)[i]	cell t c, Thread *t, int i	温度梯度矢量的分量

C_T_RG(c,t)	cell t c, Thread *t	改造后的温度梯度矢量
C_T_RG(c,t)[i]	cell t c, Thread *t, int i	改造后的温度梯度矢量的分量
C_T_M1(c,t)	cell t c, Thread *t	温度的前一次步长
C_T_M2(c,t)	cell t c, Thread *t	温度的前二次步长
C_P(c,t)	cell t c, Thread *t	压力
** C_DP in sg mem.h		
C_DP(c,t)	cell t c, Thread *t	压力梯度矢量
C_DP(c,t)[i]	cell t c, Thread *t, int I	压力梯度矢量的分量
C_U(c,t)	cell t c, Thread *t	u 方向的速度
C _V(c,t)	cell t c, Thread *t	*v*方向的速度
C_W(c,t)	cell t c, Thread *t	w方向的速度
C_H(c,t)	cell t c, Thread *t	焓
C_YI(c,t,i)	cell t c, Thread *t, int i	物质质量分数
C_K(c,t)	cell t c, Thread *t	湍流运动能
C_D(c,t)	cell t c, Thread *t	湍流运动能的分散速率
C_O(c,t)	cell t c, Thread *t	确定的分散速率

Accessing Gradient (G) Vectors and Components

You can add the the G suffix to your macro to access the gradient (G) vector of a variable quantity. For example,

C_T_G(c,t); /* returns the cell temperature gradient vector. */

returns the temperature gradient vector. Note that gradient variables are available only when the equation for that variable is being solved. For example, if you are defining a source term for energy, your UDF can access the cell temperature gradient (using C T G), but it can't get access to the x-velocity gradient (using C U G). Therefore, if you are setting up a user-defined scalar transport equation, some gradients may *not* be available. This is because the solver continually removes from memory data that it doesn't need. You can prevent the solver from freeing up memory by issuing the text command solve/set/expert, and then answering yes to the question, Keep temporary solver memory from being freed?. If you do this, all of the gradient data are retained, but

the calculation requires more memory to run.

读写梯度矢量和其分量
你可以在你的宏中加入下标_G来得到梯度矢量和它的分量，例如：
C_T_G(c,t); /* 返回单元格的温度梯度矢量 */
注意只有当已经求解出包含这个变量的方程时才能得到梯度变量。例如如果你定义了一个关于能量的原程序，那么你的UDF可以读写单元格的温度梯度（使用C_T_G）
但是你却不能读写X方向的速度分量（使用C_U_G）。而且，如果你建立了一个由使用者确定的方式转移方程，那么你就不能得到一部分的梯度了。这是因为求解器不断的移走它不需要的数据。你可以使用下面的方法来阻止存储器释放记忆：发出文本命令save/set/expert,然后对计算机提出的"是否阻止暂时的求解器记忆释放"这一提问回答"是"。按照这种做法就可以保留所有的梯度数据，但是这种计算需要更多的内存。

You can access a component of a gradient vector by specifying it as an argument in the gradient vector call (0 for the *x* component; 1 for *y*; and 2 for *z*). For example,
C_T_G(c,t)[0]; /* returns the x-component of the cell temperature */
/* gradient vector */
returns the *x* component of the temperature gradient vector. Note that in Table 5.2.1, the gradient and component macros are shown only for temperature but can be extended to all of the variables except pressure. You must use C DP to access the cell pressure gradient vector and components, as shown in Table 5.2.1.

在调用梯度矢量时把某一分量作为参数，这样就可以得到梯度分量了，（参数0代表X方向的分量，1代表Y方向的分量，2代表Z方向的分量）例如：
C_T_G(c,t)[0]; /* 返回温度梯度X方向的分量 */
注意在表5.2.1中虽然只列出了温度梯度和其分量求解的宏，但是却可以扩展到除了压力以外的所有变量中去，对于压力你只能按照表5.2.1中的方法使用C_DP来得到压力梯度和其分量。

Accessing Reconstruction Gradient (RG) Vectors and Components

In the same way as the gradient, you can add the RG su_x to your macro to access the reconstruction gradient (RG) vector and components of a variable quantity. By specifying the appropriate integer (0 for the *x* component; 1 for *y*; and 2 for *z*) as an argument, you can obtain the desired vector component. Reconstruction gradients are typically used when you want to implement your own interpolation scheme. Reconstruction gradient and component macros are shown only for temperature in Table 5.2.1, but can be extended to all of the variables. Note that RG variables, like gradient variables, are available only when the equation for that variable is being solved.

读写改造过的梯度矢量和其分量

和梯度一样的方式，你可以通过加RG的下标在你的宏中来得到梯度向量和其分量。通过使用恰当的整数作为参数来获得想要的矢量分量。（参数0代表X方向的分量，1代表Y方向的分量，2代表Z方向的分量）当你完成你自己的插补计划时可以使用改造过的梯度。改造过的温度梯度和其分量在表5.2.1中列出了，但是可以推广到所有的变量。注意改造过的梯度矢量和梯度矢量一样都只有在梯度方程被求解出来时才可以得到。

Accessing Previous Time Steps

The M1 suffix can be generally applied to the macros in Table 5.2.1 to allow access to the value of the variable at the previous time step (i.e., *t* - _*t*). These data may be useful in unsteady simulations. For example,

C_T_M1(c,t); /* returns value of cell temperature at previous time step*/

returns the value of the cell temperature at the previous time step.

The M2 suffix can be generally applied to the macros in Table 5.2.1 to allow access to the value of the variable at the time step before the previous one (i.e., *t* - 2_*t*). These data may be useful in unsteady simulations. Previous time step macros are shown only for temperature in Table 5.2.1, but can be extended to all of the variables.

读写前一步长下的时间

在表5.2.1里的宏中加入下标_M1就可以得到前一次步长时间下（t-⊿t）的变量的值。得到的这些数据可以在不稳定的模拟中使用。例如：

C_T_M1(c,t); /*返回前一步时间下的单元格温度的值 */

若在表5.2.1里宏的后面加上下标M2就可以得到前两次步长下的时间（t-2⊿t）。这些数据可用与不稳定的模拟计算中。在表5.2.1中仅列出了温度的前一次步长的求法，但是可以扩展到其它的变量中去。

5.2.2 Macros for Accessing Derivatives

The macros listed in Table 5.2.2 can be used to access velocity derivative variables.

Table 5.2.2: Derivative Macros in mem.h

5.2.2读写导数的宏

在表5.2.2中列出来的宏可以用于读写由速度导数。

表5.2.2

名称（参数）	参数类型	返回值
C DUDX(c,t)	cell t c, Thread *t	velocity derivative
C DUDY(c,t)	cell t c, Thread *t	velocity derivative
C DUDZ(c,t)	cell t c, Thread *t	velocity derivative
C DVDX(c,t)	cell t c, Thread *t	velocity derivative
C DVDY(c,t)	cell t c, Thread *t	velocity derivative
C DVDZ(c,t)	cell t c, Thread *t	velocity derivative
C DWDX(c,t)	cell t c, Thread *t	velocity derivative

C DWDY(c,t)	cell t c, Thread *t	velocity derivative
C DWDZ(c,t)	cell t c, Thread *t	velocity derivative

5.2.3 Macros for Accessing Material Properties

The macros listed in Table 5.2.3 can be used to access material property variables.

5.2.3存取材料性质的宏
在表5.2.3中列出的宏可以用于存取材料的性质。

表5.2.3在mem.h中存取材料性质的宏

名称（参数）	参数类型	返回值
C_FMEAN(c,t)	cell t c, Thread *t	第一次混合分数的平均值
C_FMEAN2(c,t)	cell t c, Thread *t	第一次混合分数的平均值
C_FVAR(c,t)	cell t c, Thread *t	第一次混合分数变量
C_FVAR2(c,t)	cell t c, Thread *t	第二次混合分数变量
C_PREMIXC(c,t)	cell t c, Thread *t	反应过程变量
C_LAM FLAME SPEED(c,t)	cell t c, Thread *t	层流焰速度
C_CRITICAL STRAIN RATE(c,t)	cell t c, Thread *t	临界应变速度
C_ POLLUT(c,t,i)	cell t c, Thread *t, int i	第i个污染物质的质量分数
C_R(c,t)	cell t c, Thread *t	密度
C_MU L(c,t)	cell t c, Thread *t	层流速度
C_MU T(c,t)	cell t c, Thread *t	湍流速度
C_MU EFF(c,t)	cell t c, Thread *t	有效粘度
C_K_L(c,t)	cell t c, Thread *t	热传导系数
C_K_T(c,t)	cell t c, Thread *t	湍流热传导系数
C_K_ EFF(c,t)	cell t c, Thread *t	有效热传导系数
C_CP(c,t)	cell t c, Thread *t	确定的热量
C_RGAS(c,t)	cell t c, Thread *t	气体常数
C_DIFF L(c,t,i,j)	cell t c, Thread *t, int i, int j	层流物质的扩散率
C_DIFF EFF(c,t,i)	cell t c, Thread *t, int i	物质的有效扩散率
C_ABS COEFF(c,t)	cell t c, Thread *t	吸附系数
C_SCAT COEFF(c,t)	cell t c, Thread *t	扩散系数
C_NUT(c,t)	cell t c, Thread *t	湍流速度for Spalart-Allmaras

5.2.4 Macros for Accessing User-Defined Scalars and Memory

The macros listed in Table 5.2.4 can be used to access user-defined scalars and memory for cells.

Table 5.2.4: User-Defined Scalar and Memory Macros for Cells in mem.h

Name(Arguments)	Argument Types	Returns
C _UDSI(c,t,i)	cell t c, Thread *t, int i	user-defined scalar (cell)
C_UDSI M(c,t,i)	cell t c, Thread *t, int i	user-defined scalar previous time step (cell)
C_UDSI_DIFF(c,t,i)	cell t c, Thread *t, int i	user-defined scalar di_usivity (cell)
C_UDMI(c,t,i)	cell t c, Thread *t, int i	user-defined memory (cell)

5.2.4用户定义的标量和存储器的宏的读写

在表5.2.4中列出的宏可以为单元格读写用户定义的标量和存储器。

表5.2.4在mem.h文件中的可以为单元格读写用户定义的标量和存储器的宏

名称（参数）	参数类型	返回值
C _UDSI(c,t,i)	cell t c, Thread *t, int i	用户定义的标量（单元格）
C_UDSI M(c,t,i)	cell t c, Thread *t, int i	前一次步长下用户定义的标量（单元格）
C_UDSI_DIFF(c,t,i)	cell t c, Thread *t, int i	用户定义的标量的分散率（单元格）
C_UDMI(c,t,i)	cell t c, Thread *t, int i	用户定义的存储器（单元格）

5.2.5 Reynolds Stress Model Macros

The macros listed in Table 5.2.5 can be used to access variables for the Reynolds stress turbulence model.

Name(Arguments)	Argument Types	Returns
C RUU(c,t)	cell t c, Thread *t	*uu* Reynolds stress
C RVV(c,t)	cell t c, Thread *t	*vv* Reynolds stress
C RWW(c,t)	cell t c, Thread *t	*ww* Reynolds stress
C RUV(c,t)	cell t c, Thread *t	*uv* Reynolds stress
C RVW(c,t)	cell t c, Thread *t	*vw* Reynolds stress
C RUW(c,t)	cell t c, Thread *t	*uw* Reynolds stress

5.2.5雷诺兹压力模型宏

在表5.2.5中列出了可以给雷诺兹压力模型读写变量的宏

表5.2.2在metric.h中的RSM宏

名字（参数）	参数类型	返回值
C RUU(c,t)	cell t c, Thread *t	*uu* 雷诺兹压力
C RVV(c,t)	cell t c, Thread *t	*vv* 雷诺兹压力
C RWW(c,t)	cell t c, Thread *t	*ww* 雷诺兹压力
C RUV(c,t)	cell t c, Thread *t	*uv*雷诺兹压力s
C RVW(c,t)	cell t c, Thread *t	*vw* 雷诺兹压力
C RUW(c,t)	cell t c, Thread *t	*uw* 雷诺兹压力

5.3 Face Macros

The macros listed in this section are defined at the boundary face of a cell and return real variables from the solver. The face variables are available in the segregated solver only. Definitions for these face macros can be found in the referenced header file (e.g., mem.h).

5.3表面宏

在这节中列出的宏是在单元格的边界面上定义的并且从求解器中返回一个真值。表面宏仅可在偏析求解器中用。这些表面宏的定义可以在相关的（.h）文件中找到。（如mem.h等）

5.3.1 Macros for Accessing Flow Variables

The macros listed in Table 5.3.1, access flow variables at a boundary face. Note that the direction of the flux determined by F_FLUX points out of the domain if the face is on the boundary.

Table 5.3.1: Macros for Accessing Flow Variables in mem.h

Name(Arguments)	Argument Types	Returns
F R(f,t)	face t f, Thread *t,	density
F P(f,t)	face t f, Thread *t,	pressure
F U(f,t)	face t f, Thread *t,	*u* velocity
F V(f,t)	face t f, Thread *t,	*v* velocity
F W(f,t)	face t f, Thread *t,	*w* velocity
F T(f,t)	face t f, Thread *t,	temperature
F H(f,t)	face t f, Thread *t,	enthalpy
F K(f t)	face t f, Thread *t,	turbulent kinetic energy
F D(f,t)	face t f, Thread *t,	turbulent kinetic energy dissipation rate
F YI(f,t,i)	face t f, Thread *t, int i	species mass fraction
F FLUX(f,t)	face t f, Thread *t	mass ow rate through a boundary face

5.3.1读写流体变量的宏

在表5.3.1中列出的宏可以在边界面读写流体变量。注意如果表面在边界上那么流体的方向是由F_FLUX决定的点指向外围空间的。

表5.3.1在mem.h中的流体变量读写的宏

名称（参数）	参数类型	返回值
F_R(f,t)	face t f, Thread *t,	密度
F_P(f,t)	face t f, Thread *t,	压力
F_U(f,t)	face t f, Thread *t,	*u*方向的速度
F_V(f,t)	face t f, Thread *t,	*v* 方向的速度
F_W(f,t)	face t f, Thread *t,	*w*方向的速度
F_T(f,t)	face t f, Thread *t,	温度
F_H(f,t)	face t f, Thread *t,	焓
F_K(f t)	face t f, Thread *t,	湍流运动能
F_D(f,t)	face t f, Thread *t,	湍流运动能的分散速率
F_YI(f,t,i)	face t f, Thread *t, int i	物质的质量分数
F_FLUX(f,t)	face t f, Thread *t	通过边界表面的质量流速

5.3.2 Macros for Accessing User-Defined Scalars and Memory

The macros listed in Table 5.3.2 can be used to access user-defined scalars and memory for faces.

Name(Arguments)	Argument Types	Returns
F_ UDSI(f,t,i)	face t f, Thread *t, int i	user-de_ned scalar (face)
F_UDMI(f,t,i)	face t f, Thread *t, int i	user-de_ned memory (face)

5.3.2读写用户定义的标量和存储器的宏

在表5.3.2中列出的宏可以用于给表面读写用户定义的标量和存储器的宏

名称（参数）	参数类型	返回值
F_UDSI(f,t,i)	face t f, Thread *t, int i	用户确定的标量（表面）
F_UDMI(f,t,i)	face t f, Thread *t, int i	用户定义的存储器（表面）

5.3.3 Miscellaneous Face Variable Macros

Additional face variable macros are listed in Table 5.3.3.

Table 5.3.3: Miscellaneous Face Variable Macros

Name(Arguments)	Argument Types	Returns
F_C0(f,t)	face t f, Thread *t	cell t for cell on \0" side of face
F_C0_THREAD(f,t)	face t f, Thread *t	Thread *t for cell thread on \0" side of face
F_C1(f,t)	face t f, Thread *t	cell t for cell on \1" side of face
F_C1_THREAD(f,t)	face t f, Thread *t	Thread *t for cell thread on \1" side of face

5.3.3混合面变量宏

其余的表面变量宏在表5.3.3中列出

名称（参数）	参数类型	返回值
F_C0(f,t)	face t f, Thread *t	
F_C0_THREAD(f,t)	face t f, Thread *t	
F_C1(f,t)	face t f, Thread *t	
F_C1_ THREAD(f,t)	face t f, Thread *t	

5.4 Geometry Macros

This section contains macros for retrieving geometry variables in FLUENT. Definitions for these macros can be found in the referenced header file (e.g., mem.h).

5.4几何宏

这节包含的宏是为了在FLUENT中重新得到几何变量。这些表面宏的定义可以在相关的（.h）文件中找到。（如mem.h等)

5.4.1 Number of Nodes and Faces

The macros C_NNODES and C_NFACES shown in Table 5.4.1 return the integer number of nodes or faces, respectively, for a given cell. F_NNODES returns the integer number of nodes associated with a face.

Table 5.4.1: Node and Face Variable Macros in mem.h

Name(Arguments)	Argument Types	Returns
C NNODES(c,t)	cell t c, Thread *t	number of nodes in a cell
C NFACES(c,t)	cell t c, Thread *t	number of faces in a cell
F NNODES(f,t)	face t f, Thread *t	number of nodes in a face

5.4.1节点和面的数量

在表5.4.1中列出的宏C_NNODES和C_NFACES返回相应的节点和面的整数值。对于一个给定的单元格， F_NNODES返回与某个表面相关的节点的整数个数。

表5.4.1在mem.h中的节点和表面的宏

名称（参数）	参数类型	返回值
C_NNODES(c,t)	cell t c, Thread *t	一个单元格中的节点数
C_NFACES(c,t)	cell t c, Thread *t	一个单元格中的表面数
F_NNODES(f,t)	face t f, Thread *t	一个表面中节点数

5.4.2 Cell and Face Centroids

The macros listed in Table 5.4.2 can be used to obtain the real centroid of a cell or face. C_CENTROID finds the coordinate position of the centroid of the cell c and stores the coordinates in the x array. F_CENTROID finds the coordinate position of the centroid

of the face f and stores the coordinates in the x array. Note that the array x can be a one-, two-, or three-dimensional array.

Table 5.4.2: Centroid Variable Macros in metric.h

Name(Arguments)	Argument Types	Returns
C CENTROID(x,c,t)	real x[ND ND], cell t c, Thread * t	x (cell centroid)
F CENTROID(x,f,t)	real x[ND ND], face t f, Thread *t	x (face centroid)

5.4.2单元格和表面的重心

在表5.4.2中列出的宏可以用来获得一个单元格或是表面的真实的重心。C_CENTROID找到单元格的重心的坐标，并把它的坐标存储在矩阵X中。F_CENTROID找到表面的重心的坐标，并把它的坐标存储在矩阵X中。注意矩阵X可以是一维，二维或者是三维的。

表5.4.2 在metric.h中变量重心宏

名字（参数）	参数类型	返回值
C_CENTROID(x,c,t)	real x[ND ND], cell t c, Thread * t	x (单元格重心)
F_CENTROID(x,f,t)	real x[ND ND], face t f, Thread *t	x (表面中心)

5.4.3 Face Area

The macro F_AREA shown in Table 5.4.3 can be used to return the real face area vector. For interior faces, the face area normal vector points from cell c0 to cell c1. The normal always points outward (out of the domain) for boundary faces.

Table 5.4.3: Face Area Macros in metric.h

Name(Arguments)	Argument Types	Returns
F_AREA(A,f,t)	A[ND ND], face t f, Thread *t	A (area vector)

5.4.3 表面积

在表5.4.3中列出的宏F_AREA可以被用于返回一个实数的面积向量。对于内部的表面，标准的面积向量的方向是从单元格C0指向单元格C1。标准的方向总是从边界面向外指（范围之外）。

表5.4.3 在metric.h中的表面积宏

名称（参数）	参数类型	返回值
F_AREA(A,f,t)	A[ND ND], face t f, Thread *t	A (面积向量)

5.4.4 Cell Volume

The macros listed in Table 5.4.4 can be used to obtain the real cell volume for 2D, 3D, and axisymmetric simulations.

Table 5.4.4: Cell Volume Macros in mem.h

Name(Arguments)	Argument Types	Returns
C VOLUME(c,t)	cell t c, Thread *t	cell volume for 2D or 3D, cell volume/2_ for axisymmetric

5.4.4单元格体积

在表5.4.4中列出的宏可以用于获得二维，三维和轴对称的模型的单元格的真实体积

表5.4.4在mem.h中的单元格体积宏

名称（参数）	参数类型	返回值
C VOLUME(c,t)	cell t c, Thread *t	二维或是三维的单元格体积 单元格体积/2π 是轴对称模型的体积

5.4.5 Cell-to-Cell and Cell-to-Face Centroid Macros

FLUENT provides macros that allow the vectors connecting cell centroids and the vectors connecting cell and face centroids to be readily defined. These macros return information that are helpful in evaluating face values of scalars which are generally not stored, as well as the diffiusive flux of scalars across cell boundaries. The geometry and gradients involved with these macros are summarized in Figure 5.4.1 below.

To better understand the parameters that are returned by these macros, it is best to consider how the aforementioned calculations are evaluated. Assuming that the gradient of a scalar is available, the face value of a scalar, _ can be approximated by,

$$\phi_f = \phi_0 + \nabla\phi . d\bar{r} \qquad (5.4.1)$$

where ~ *dr* is the vector that connects the cell centroid with the face centroid. The gradient in this case is evaluated at the cell centroid where _0 is also stored.

where _f is the di_usion coe_cient at the face. In FLUENT's unstructured solver, the gradient along the face normal direction may be approximated by evaluating gradients along the directions that connect cell centroids and along a direction confined within the plane of the face. Given this, D_f may be approximated as,

$$D_f = \Gamma_f \nabla\phi \vec{A} \qquad (5.4.2)$$

where _f is the di_usion coe_cient at the face. In FLUENT's unstructured solver, the gradient along the face normal direction may be approximated by evaluating gradients along the directions that connect cell centroids and along a direction con_ned within the plane of the face. Given this, D_f may be approximated as,

$$D_f = \Gamma_f \frac{(\phi_1 - \phi_0)}{ds} \frac{\vec{A}\cdot\vec{A}}{\vec{A}\cdot\vec{e}_s} + \Gamma_f (\nabla\phi\cdot\vec{A} - \nabla\phi\cdot\vec{e}_s \frac{\vec{A}\cdot\vec{A}}{\vec{A}\cdot\vec{e}_s}) \qquad (5.4\text{-}3)$$

where the _rst term on the right hand side represents the primary gradient directed along the vector ~e_s and the second term represents the 'cross' di_usion term. In this equation, *A* is the area normal vector of face *f* directed from cell c0 to c1, *ds* is the distance between the cell centroids, and ~e_s is the unit normal vector in this direction. This is shown in Figure 5.4.1 below.

5.4.5单元格对单元格，单元格对表面重心宏

FLUENT提供的宏使得连接单元格重心的向量和连接单元格重心与表面重心的向量很容易定义。这些宏的返回信息对于估计那些不储存的表面的数量值是有帮助的，同时对于估计通过单元格边界的流体分散率的数值也有益。这些宏所包含的几何和梯度在下面的表5.4.1中总结出来了。

为了更好的理解由这些宏所返回的参数，最好的方法是考虑一下上述计算是如何估计的。假定可以得到一个向量的梯度可以得到，那么向量的表面值φ可以用下式估计：

$$\phi_f = \phi_0 + \nabla\phi . \bar{d} r \qquad (5.4.1)$$

在这里$\bar{d}r$是连接单元格重心和表面重心的向量。在这个例子中是在存储ϕ_0处的单元格重心位置估计的。

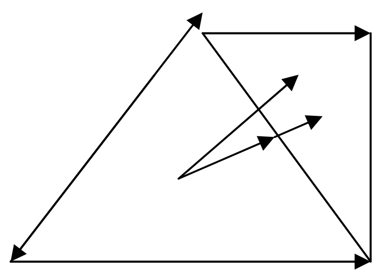
通过表面f，向量ϕ的流体分散率D_f由下式给定

$$D_f = \Gamma_f \nabla\phi\vec{A} \qquad (5.4.2)$$

这的Γ_f是表面的分散系数。在FLUENT 的未定结构求解器中，沿表面标准方向的梯度可以由在一个沿单元格重心方向梯度和一个限定在表面的平面里的梯度方向来估计。按照这种规定，D_f可以按下式近似：

$$D_f = \Gamma_f \frac{(\phi_1 - \phi_0)}{ds} \frac{\vec{A}\cdot\vec{A}}{\vec{A}\cdot\vec{e}_s} + \Gamma_f (\nabla\phi\cdot\vec{A} - \nabla\phi\cdot\vec{e}_s \frac{\vec{A}\cdot\vec{A}}{\vec{A}\cdot\vec{e}_s}) \qquad (5.4\text{-}3)$$

在此右边的第一项代表沿着向量$\vec{e}_s$方向的主梯度，第二项代表交叉分散率项。在这个方程中A是从单元格c0指向单元格c1的表面f的标准向量围成的面积，ds是两个单元格重心间的距离，$\vec{e}_s$是这个方向的单元向量。这些在下面的图形5.4.1中表示出来了



图形 5.4.1 用定义的梯度和向量连接单元格 c0 和 c1

The INTERIOR_FACE_GEOMETRY and BOUNDARY_FACE_GEOMETRY macros can be called to return some of the terms needed to evaluate Equations 5.4-1 and 5.4-3.
调用宏 INTERIOR_FACE_GEOMETRY 和 BOUNDARY_FACE_GEOMETRY 可以返回一些可以估计方程 5.4_1 和 5.4_2 的一项。

INTERIOR_FACE_GEOMETRY

INTERIOR_FACE_GEOMETRY(f,tf,A,ds,es,A_by_es,dr0,dr1) returns for the face, f, in face thread tf, the following variables:

A[ND_ND] 面积的单元向量
ds 单元格重心的距离
es[ND_ND] 从单元格c0到单元格c1方向的单元向量
A_by_es $\frac{\vec{A}\cdot\vec{A}}{\vec{A}\cdot\vec{e}_s}$ 的值
dr0[ND_ND] 连接单元格c0重心和表面重心的单元向量
dr1[ND-ND] 连接单元格c1重心和表面重心的单元向量

Note that the INTERIOR_FACE_GEOMETRY macro is defined in the sg.h header file. Since sg.h is not included in the udf.h file, you will need to add the include directive to your UDF.
注意宏INTERIOR_FACE_GEOMETRY在文件sg.h中定义，因为sg.h没有包含在udf.h文件中，所以你就必须把它加到你的UDF中。

BOUNDARY FACE GEOMETRY(f,tf,A,ds,es,A by es,dr0) returns for the face, f, in face thread tf, the following variables:
A[ND_ND] 面积标准向量
ds 单元格重心和表面重心之间的距离
es[ND_ND] 从单元格c0重心到表面重心的单元向量
A_by_es $\frac{\vec{A}\cdot\vec{A}}{\vec{A}\cdot\vec{e}_x}$ 的值
dr0[ND_ND] 连接单元格c0重心到表面重心的向量

The BOUNDARY_FACE_GEOMETRY 宏也被定义在sg.h文件中，它也没有被包含在udf.h中，你需要包含sg.h在UDF。

5.5 Node Macros

The macros listed in Tables 5.5.1 and 5.5.2 return the real Cartesian coordinates of the cell node (at the cell corner), and the components of the velocity of the node, respectively. The node velocities are relevant in moving mesh simulations, for example. The argument Node *node for each of the variables defines a node. Definitions for these macros can be found in the referenced header file (e.g., mem.h).

Table 5.5.1: Node Coordinate Variable Macros in metric.h

Name(Arguments)	Argument Types	Returns
NODE X(node)	Node *node	*x* coordinate of node
NODE Y(node)	Node *node	*y* coordinate of node
NODE Z(node)	Node *node	*z* coordinate of node

Table 5.5.2: Node Velocity Variable Macros in mem.h

Name(Arguments)	Argument Types	Returns
NODE GX(node)	Node *node	*x* component of node velocity
NODE GY(node)	Node *node	*y* component of node velocity
NODE GZ(node)	Node *node	*z* component of node velocity

5.5节点宏

表5.5.1和5.5.2列出的宏返回单元格节点的实数直角坐标（在单元格的拐角）和相应的节点速度的分量。例如在移动的网格模拟中节点速度是相对应的。每个变量的节点*节点的参数定义了一个节点。这些宏的定义可以在相关的.h文件中找到。（例如mem.h）

表5.5.1在 metric.h 中变量的节点坐标宏

名称（参数）	参数类型	返回值
NODE X(node)	Node *node	节点的X坐标
NODE Y(node)	Node *node	节点的Y坐标
NODE Z(node)	Node *node	节点的Z坐标

表5.5.2在 metric.h中的节点速度变量宏

名称（参数）	参数类型	返回值
NODE GX(node)	Node *node	节点速度的X分量
NODE GY(node)	Node *node	节点速度的Y分量
NODE GZ(node)	Node *node	节点速度的Z分量

5.6 Multiphase Macros

The macros listed in Table 5.6.1 return real variables associated with the general multiphase models. Definitions for these macros can be found in sg_mphase.h, which is included in udf.h.

Table 5.6.1: Multiphase Variable Macros in sg_mphase.h

Name(Arguments)	Argument Types	Returns
C VOF(c,pt[0])	cell t c, Thread **pt	volume fraction for primary phase

C VOF(c,pt[n])	cell t c, Thread **pt	volume fraction for *n*th secondary phase

5.6多相宏

表5.6.1中列出的宏返回一个与整体多相节点相连的实数变量。这些变量的定义在sg_mphase.h文件中可以找到，这些包含在udf.h.文件中。

表5.6.1在sg_mphase.h中的变量宏

名称（参数）	参数类型	返回值
C VOF(c,pt[0])	cell t c, Thread **pt	主要相的体积分数
C VOF(c,pt[n])	cell t c, Thread **pt	第n个辅助相的体积分数

5.7 DPM Macros

The macros listed in Tables 5.7.1{5.7.4 are defined in the dpm.h header file, which is included in the udf.h header file.

The variable p indicates a pointer to the Tracked Particle structure (Tracked Particle *p). Tracked particle *p gives you the value for the particle at the current position..

Name(Arguments)	Argument Types	Returns
P_DIAM(p)	Tracked particle *p	particle diameter
P_VEL(p)[i]	Tracked particle *p	particle velocity i=0,1,2
P-T(p)	Tracked particle *p	particle temperature
P-RHO(p)	Tracked particle *p	particle density
P-MASS(p)	Tracked particle *p	particle mass
P-TIME(p)	Tracked particle *p	current time for particle
P-DT(p)	Tracked particle *p	particle time step
P_LF(p)	Tracked particle *p	particle liquid fraction (wet combusting particles only)
P-VFF(p)	Tracked particle *p	particle volatile fraction (combusting particles only)

5.7 DPM宏

在表5.7.1—5.7.4中列出的宏是在dpm.h文件中定义的这些都是包含在udf.h文件中的。

变量p是Tracked_Particle结构的指示器（Tracked_Particle*p）。Tracked_Particle*p给出物体在当前位置的值。

表5.7.1中在dpm.h文件中在质点在当前位置的宏

Name(Arguments)	Argument Types	Returns
P_DIAM(p)	Tracked particle *p	质点直径
P_VEL(p)[i]	Tracked particle *p	质点速度 i=0,1,2
P_T(p)	Tracked particle *p	质点温度
P_RHO(p)	Tracked particle *p	质点密度

P_MASS(p)	Tracked particle *p	质点质量
P_TIME(p)	Tracked particle *p	质点的当前时间
P_DT(p)	Tracked particle *p	质点时间步长
P_LF(p)	Tracked particle *p	质点液体分数（湿的组分燃烧的质点）
P_VFF(p)	Tracked particle *p	质点挥发性馏分（仅仅燃烧的质点）

Table 5.7.4: Macros for Particle Material Properties in dpm.h

Name(Arguments)	Argument Types	Returns
P_MATERIAL(p)	Tracked particle *p	material pointer
DPM-SWELLING-COEFF(p)	Tracked particle *p	swelling coefficient for devolatilization
DPM-EMISSIVITY(p)	Tracked particle *p	emissivity for radiation model
DPM-SCATT-FACTOR(p)	Tracked particle *p	scattering factor for radiation model
DPM-EVAPORATION TEMPERATURE(p)	Tracked particle *p	evaporation temperature
DPM-BOILING-TEMPERATURE(p)	Tracked particle *p	boiling temperature
DPM-LATENT HEAT(p)	Tracked particle *p	latent heat
DPM_HEAT OF PYROLYSIS(p)	Tracked particle *p	heat of pyrolysis
DPM_HEAT OF REACTION(p)	Tracked particle *p	heat of reaction
DPM_VOLATILE FRACTION(p)	Tracked particle *p	volatile fraction
DPM-CHAR-FRACTION(p)	Tracked particle *p	char fraction
DPM_SPECIFIC -HEAT(p,t)	Tracked particle *p	specific heat at temperature t

表5.7.4在dpm.h中描述质点材料性质的宏

名称（参数）	参数类型	返回值
P_MATERIAL(p)	Tracked particle *p	材料的指示器
DPM_SWELLING-COEFF(p)	Tracked particle *p	液化作用的膨胀系数
DPM-EMISSIVITY(p)	Tracked particle *p	辐射模型的分散系数
DPM –SCATT-FACTOR(p)	Tracked particle *p	辐射模型的分散因子
DPM-EVAPORATION TEMPERATURE(p)	Tracked particle *p	蒸发温度
DPM-BOILING-TEMPERATURE(p)	Tracked particle *p	沸点温度
DPM-LATENT-HEAT(p)	Tracked particle *p	潜热
DPM-HEAT OF PYROLYSIS(p)	Tracked particle *p	高温分解热
DPM-HEAT OF REACTION(p)	Tracked particle *p	反应热
DPM-VOLATILE FRACTION(p)	Tracked particle *p	挥发性馏分
DPM-CHAR-FRACTION(p)	Tracked particle *p	烧焦分数
DPM-SPECIFIC-HEAT(p,t)	Tracked particle *p	在温度t下的确定热

5.8 NO_x Macros

The macros listed in Table 5.8.1 are useful for defining NO_x production and reduction rates using DEFINE_NOX_RATE. These macros are defined in the sg_nox.h header file, which is included in udf.h.

The argument NOx is the pointer to the NO_x structure.

Table 5.8.1: Macros for NO_x Rates in sg_nox.h

Name(Arguments)	Argument Types	Returns
NOX EQN(nox)	NOx *nox	NO_x pollutant equation ID
NOX FRATE(nox)	NOx *nox	NO_x production rate
NOX RRATE(nox)	NOx *nox	NO_x reduction rate
ARRH(nox, k)	NOx *nox, Rate const k[3]	Arrhenius rate coefficient
MOLECON(nox, i)	NOx *nox, int *i*	species *i* molar concentration

In the argument Rate const k[3],

k = A, β , E

where,

A = Pre-exponential factor

β = Temperature Exponent

E = Activation Energy

5.8 Nox宏

在表5.8.1中列出的宏对于使用DEFINE_NOX_RATE来定义No_x的产生和减少是有用的。这些宏定义在sg_nox.h中，它包含在udf.h里。

参数No_x是No_x结构的指示器。

表5.8.1 在sg_nox.h中描述No_x速率的宏

名称（参数）	参数类型	返回值
NOX EQN(nox)	NOx *nox	No_x污染物质方程 ID
NOX FRATE(nox)	NOx *nox	NO_x 产生速率
NOX RRATE(nox)	NOx *nox	No_x减少速率
ARRH(nox, k)	NOx *nox, Rate const k[3]	阿列纽斯速率系数

MOLECON(nox, i)	NOx *nox, int *i*	物质 *I*的摩尔浓度

在参数Rate_ const k[3] 中,

k = A, β, E

这里

A =指前因子

β =温度指数

E = 活化能β

6. Utilities （工具）

本章提供了针对 FLUENT 变量性能计算的 FLUENT 公司提供的预处理工具列表。

- 6.1 Introduction 简要
- 6.2 General-Purpose Looping Macros
- 一般目的的宏
- 6.3 Multiphase-Specific Looping Macros
- 多项组分的宏
- 6.4 Setting Face Variables (F_PROFILE) 设置面变量
- 6.5 Accessing Variables That Are Not Passed as Arguments
- 访问没有作为 Argument 传递的变量
- 6.6 Accessing Neighboring Cell and Thread Variables

  访问邻近单元（网格点和线）上的变量

- 6.7 User-Defined Memory for Cells (C_UDMI)

  用户为网格定义内存（C－UDMI）

- 6.8 Vector Utilities

  矢量工具

- 6.9 Macros for Time-Dependent Simulations

  与时间相关的数值模拟宏

- 6.10 Miscellaneous Utilities

  其他各种工具

6.1 简要

Fluent Inc.提供了针对 Fluent 变量操作的一系列工具。这些工具中大部分可以作为宏直接执行。

许多宏可以用于单相和多相模型的 UDFs 中，但是有些是只针对多相流的。回想一下当你为多相流模型写 UDFs 时，你将记住 FLUENT 的结构的层次。（详细参考 3.11.1）。

从求解器中导入到你的 UDFs 中特殊的控制区和线性结构，依赖于你所使用的 DEFINE 宏和相关的控制区函数。（通过图形用户界面和用户定义的源代码）

它或许也依赖于你所使用的多相流模型。将控制区的结构传递给 DEFINE_INIT 和 DEFINE_ADJUST 函数,但是它与多相流模型是相互独立的。这些函数始终被传递给与混合物有关的控制区结构。DEFINE_ON_DEMAND UDFs 没有被传递给任何控制区。

如果你的 PDF 没有显式地传给你的函数所需要的线性的或者控制区的结构，那么你可以利用本章提供的宏工具修复。提供的许多宏使你的函数可以寻找到给定线和区的所有的网格点和面。

6.2 一般目的的循环宏

下面这些循环的宏可以用于 FLUENT 单相和多相模型的 UDFs 中。这些宏的定义包含再 mem.h 头文件中。

- 6.2.1 Looping over Cell Threads in a Domain (thread_loop_c)
- 查询控制区的单元线
- 6.2.2 Looping over Face Threads in a Domain (thread_loop_f)
- 查询控制区的面
- 6.2.3 Looping over Cells in a Cell Thread (begin...end_c_loop)
- 查询单元线中的单元
- 6.2.4 Looping over Faces in a Face Thread (begin...end_f_loop)
- 查询面单元中的面
- 6.2.5 Looping over Faces on a Cell (c_face_loop)
- 查询单元面
- 6.2.6 Looping over Nodes of a Cell (c_node_loop)
- 查询单元节点

6．2.1 查询控制区的单元线

当你想查询给定控制区的单元线时，你可以用 thread_loop_c。它包含单独的说明，后面是对控制区的单元线所做操作，正如下面显示的包含在{ }中。注意：thread_loop_c 在执行上和 thread_loop_f 相似，参考 6.2.2 部分。

```
Domain *domain;
Thread *c_thread;
thread_loop_c(c_thread, domain) /*loops over all cell threads in domain*/
  {
  }
```

6.2.2 查询控制区的面

当你想要查询给定控制区的面时，你可以应用 thread_loop_f。它包含单独的说明，后面是对控制区的面单元所做操作，正如下面显示的包含在{ }中。注意：thread_loop_f 在执行上和 thread_loop_c 相似，参考 6.2.1 部分。

```
Thread *f_thread;
Domain *domain;
thread_loop_f(f_thread, domain)/* loops over all face threads in a
domain*/
  {

  }
```

6.2.3 查询单元线中的单元

当你想要查询给定单元线 c_thread 上所有的单元时，你可以应用 begin_c_loop 和 end_c_loop。它包含 begin 和 end_loop 的说明，完成对单元线中单元所做的操作，定义包含在{ }中。当你想查找控制区单元线的单元时，应用的 loop 全嵌套在 thread_loop_c 中。

```
cell_t c;
Thread *c_thread;
begin_c_loop(c, c_thread)    /* loops over cells in a cell thread  */
  {
  }
end_c_loop(c, c_thread)
```

例子：

```
/* Loop over cells in a thread to get information stored in cells. */
  begin_c_loop(c, c_thread)
    {
     /* C_T gets cell temperature. The += will cause all of the cell
        temperatures to be added together. */

      temp += C_T(c, c_thread);
    }
```

```
  end_c_loop(c, c_thread)
}
```

6.2.4 查询面线中的面

当你想要查找给定面线 f_thread 的所有的面时，你可以用 begin_f_loop and end_f_loop。它包含 begin 和 end loop 的说明，完成对面线中面单元所做的操作，定义包含在{}中。当你想查找控制区面线的所有面时，应用的 loop 全嵌套在 thread_loop_f 中。

```
face_t f;
Thread *f_thread;
begin_f_loop(f, f_thread)     /* loops over faces in a face thread  */
  {
  }
end_f_loop(f, f_thread)
```

例子：

```
/* Loop over faces in a face thread to get the information stored on faces.
*/
  begin_f_loop(f, f_thread)
    {
      /*  F_T gets face temperature. The += will cause all of the face
          temperatures to be added together. */

      temp += F_T(f, f_thread);
    }
  end_f_loop(f, f_thread)
```

6.2.5 查询单元中的面

下面函数用以查询给定单元中所有面。包含单独的查询说明，后面是所做的操作包含在{}。

```
face_t f;
Thread *tf;
int n;
c_face_loop(c, t, n)          /* loops over all faces on a cell */
  {
  .
```

```
  .
  .
  f = C_FACE(c,t,n);
  tf = C_FACE_THREAD(c,t,n);
  .
  .
  .
  }
```

这里，n 是当地面的索引号。当地面的索引号用在 C_FACE 宏中以获得所有面的数量(e.g., f = C_FACE(c,t,n))。

另一个在 c_face_loop 中有用的宏是 C_FACE_THREAD。这个宏用于合并两个面线。

(e.g., tf = C_FACE_THREAD(c,t,n)).

查找与 c_face_loop 有关的宏参考 6.10 部分。

6.2.6 查询单元节点(c_node_loop)

下面函数用以查询给定单元中所有节点。包含单独的查询说明，后面是所做的操作包含在{}。

```
cell_t c;

Thread *t;
int n;
c_node_loop(c, t, n)
  {
  .
  .
  .
  node = C_NODE(c,t,n);
  .
  .
  .
  }
```

这里，n 是当地节点的索引号。当地面的索引号用在 C_NODE 宏中以获得所有面的数量((e.g., node = C_NODE(c,t,n)

6.3 多相组分查询宏

下面这些宏用于多相模型的 UDFs。关于 FLUENT 里的结构的层次的讨论参考 3.11 部分尤其是图 3.11.1。

- 6.3.1 Looping over Phase Domains in a Mixture (sub_domain_loop)
- 查询混合物中的相控制区
- 6.3.2 Looping over Phase Threads in a Mixture (sub_thread_loop)
- 查询混合物中的相线
- 6.3.3 Looping over Phase Cell Threads in a Mixture (mp_thread_loop_c)
- 查询混合物中的相单元线
- 6.3.4 Looping over Phase Face Threads in a Mixture (mp_thread_loop_f)
- 查询混合物中的相的面线

6.3.1 Looping over Phase Domains in a Mixture

6.3.1 查询混合物中相的控制区(sub_domain_loop)

sub_domain_loop 宏用于查询混合物控制区的所有相的子区。这个宏查询并在混合物控制区给每个相区定义指针以及相关的 phase_domain_index。正如 3.11 部分所讨论的，控制区需要指针，在每个相中都有权访问部分数据。注意：sub_domain_loop 宏在执行中和 sub_thread_loop 宏是相似的,参考 6.3.2 部分。

```
int phase_domain_index;    */ index of subdomain pointers  */
Domain *mixture_domain;
Domain *subdomain;
sub_domain_loop(subdomain, mixture_domain, phase_domain_index)
```

sub_domain_loop 的变量是 subdomain,mixture_domain 和 phase_domain_index。

Subdomain 是 phase-level domain 的指针， mixture_domain 是 mixture-level domain 的指针。当你想用 DEFINE 宏时，mixture_domain（包含控制区变量 e.g., DEFINE_ADJUST）通过 FLUENT 求解器自动传递给你的 UDF，混合物就和你的 UDF 相关了。如果

mixture_domain 没有显式地传递给你的 UDF,你应用另外一个宏来恢复它 (e.g., Get_Domain(1) before calling sub_domain_loop（参考 6.5.1 部分）。phase_domain_index 是子区指针所引号， phase_domain_index 是初始相地索

引号为 0，混合物中其它相依次加 1。注意：subdomain 和 phase_domain_index 是在 sub_domain_loop 宏定义中初始化的。

例子：

下列被集成在 UDF 中的语句在求解过程中补充说明一个相的体积分数。它在求解过程的开始执行。这个函数建立一个中心在 0.5, 0.5, 0.5，半径为 0.25 的球形体。第二个相的体积分数 1 被补充说明到球形体内的单元中，但是第二个相在其他单元中的体积分数为 0。

```
/*****************************************************************/
/* UDF for initializing phase volume fraction                    */
/*****************************************************************/

#include "udf.h"

/* domain pointer that is passed by INIT function is mixture domain  */
DEFINE_INIT(my_init_function, mixture_domain)
{
  int phase_domain_index;
  cell_t cell;
  Thread *cell_thread;
  Domain *subdomain;
  real xc[ND_ND];

  /* loop over all subdomains (phases) in the superdomain (mixture) */
  sub_domain_loop(subdomain, mixture_domain, phase_domain_index)
  {
     /* loop if secondary phase  */
     if (DOMAIN_ID(subdomain) == 2)

     /* loop over all cell threads in the secondary phase domain  */
     thread_loop_c (cell_thread,subdomain)
     {

         /* loop over all cells in secondary phase cell threads  */
         begin_c_loop_all (cell,cell_thread)
         {
             C_CENTROID(xc,cell,cell_thread);
             if (sqrt(ND_SUM(pow(xc[0] - 0.5,2.),
                             pow(xc[1] - 0.5,2.),
                             pow(xc[2] - 0.5,2.))) < 0.25)

                /*  set volume fraction to 1 for centroid  */
```

```
            C_VOF(cell,cell_thread) = 1.;

          else
            /*  otherwise initialize to zero  */
            C_VOF(cell,cell_thread) = 0.;
        }
        end_c_loop_all (cell,cell_thread)
    }

  }
}
```

6.3.2 查询混合物的相线 (sub_thread_loop)

sub_thread_loop 宏查询所有与混合物等值线相的等值线。这个宏查找并返回每个子区和相关 phase_domain_index 的指针。

如 3.11 部分所讨论的，如果 subthread 指针 与进口区域相关，那么这个宏将提供给进口区域每个相面线指针。

```
int phase_domain_index;

Thread *subthread;
Thread *mixture_thread;
sub_thread_loop(subthread, mixture_thread, phase_domain_index)
```

sub_thread_loop 的自变量是 subthread, mixture_thread, 和 phase_domain_index。 subthread 是相线的指针，mixture_thread 是 mixture-level thread 的指针。当你用 DEFINE 宏（包含一个线自变量）时，通过 FLUENT 的求解器 mixture_thread 自动传递给你的 UDF，你的 UDF 就和混合物相关了。如果 mixture_thread 没有显式地传递给你的 UDF，你需要在调用 sub_thread_loop 之前，调用工具宏恢复它。phase_domain_index 是子区指针索引号，可以用宏 PHASE_DOMAIN_INDEX 恢复。（详情参考 6.5.9 部分）初始相的索引号为 0，混合物中其它相依次加一。注意： subthread 和 phase_domain_index 在 sub_thread_loop 宏定义中被初始化。

6.3.3 查询混合物中所有单元的线(mp_thread_loop_c)

mp_thread_loop_c 宏查询混合物控制区所有单元的线，提供了与混合物等值线有关的相等值线的指针。当应用到混合物控制区时这几乎和 thread_loop_c 宏是等价的。（ 6.2.1 部分）区别是：除了查询每个单元线 ，它还返回一个指针数组(pt）等价与相等值线。单元线第 i 相的指针是 pt[i]，这里 i 是相控制区索引号 phase_domain_index。pt[i] 可以用做宏的自变量。需要相等值线的指针。相控制区索引号 phase_domain_index 可以用宏 PHASE_DOMAIN_INDEX 恢复(详情参考 6.5.9 部分)。

```
Thread **pt;
Thread *cell_threads;
Domain *mixture_domain;
mp_thread_loop_c(cell_threads, mixture_domain, pt)
```

mp_thread_loop_c的自变量是cell_threads, mixture_domain ， pt. cell_threads 是网格线的指针， mixture_domain 是mixture-level控制区的指针，. pt 是含有phase-level 线的指针数组。

当你要用包含控制区变量(e.g., DEFINE_ADJUST 的宏 DEFINE 时，mixture_domain 通过 FLUENT 的求解器自动传递给你的 UDF 文件,你的 UDF 就和混合物相关了。若 mixture_domain 没有显式地传递给你 UDF 文件，你应用另外一个工具(e.g., Get_Domain(1)来恢复，参考 6.5.1 部分。注意： pt 和 cell_threads 的值是由查询函数派生出来的。mp_thread_loop_c 是典型的用于 begin_c_loop 中。 begin_c_loop 查询网格线内的所有网格。当 begin_c_loop 嵌套在 mp_thread_loop_c 中,你就可以查询混合物中相单元线的所有网格了。

6.3.4 查询混合物中所有的相面线(mp_thread_loop_f)

宏 mp_thread_loop_f 查询混合物控制区内所有混合物等值线的面线并且给每个与混合物等值线有关的相等值线指针。在混合物控制区内这和宏 thread_loop_f 几乎是等价的。区别是：除了查找每一个面线，这个宏还返回一个指针数组 pt，它与相等值线相互关联。 指向第 i 相的面线指针是 pt[i]，这里是 phase_domain_index。 当需要相等值线指针时，pt[i] 可以作为宏的自变量。phase_domain_index 可以用宏 PHASE_DOMAIN_INDEX 恢复。（参考 6.5.9)

```
Thread **pt;
Thread *face_threads;
Domain *mixture_domain;
mp_thread_loop_f(face_threads, mixture_domain, pt)
```

mp_thread_loop_f 的自变量是 face_threads, mixture_domain, 和 pt。face_threads 是面线的指针，是混合物等值线控制区的指针。pt 是包含相等值线的指针数组。

当你要用包含控制区变量(e.g., DEFINE_ADJUST 的宏 DEFINE 时，mixture_domain 通过 FLUENT 的求解器自动传递给你的 UDF 文件,你的 UDF 就和混合物相关了。若 mixture_domain 没有显式地传递给你 UDF 文件，你应用另外一个工具(e.g., Get_Domain(1)来恢复，参考 6.5.1 部分。注意： pt 和 cell_threads 的值是由查询函数派生出来的。mp_thread_loop_f 是典型的用于 begin_f_loop 中。 begin_f_loop 查询网格线内的所有网格。当 begin_f_loop 嵌套在 mp_thread_loop_f 中,你就可以查询混合物中相单元线的所有网格了。

6.4 Setting Face Variables (F_PROFILE)

6.4 设置面变量

当你要设置面的变量的值时，应用F_PROFILE宏。当你要生成边界条件的外形或存储新的变量值时，自动调用这一函数。F_PROFILE全部宏定义参考mem.h文件。.

宏: F_PROFILE(f, t, n)

自变量: face_t f

Thread *t

int n

函数返回值: void

F_PROFILE 的自变量是 f, 面的索引号是 face_t; t, 面上线的指针,还有一个整数 n。 这些变量通过 FLUENT 的求解器自动传递给你的 UDF。你不必给他们赋值。 整数 n 是要在边界上设定的变量标志符。例如：进口边界包含总压和总温，二者都在用户定义函数中定义。 进口边界的变量在 FLUENT 赋予整数 0，其它赋予 1。当你在 FLUENT 的进口边界面板中定义边界条件时，这些整数值由求解器设定。

6.5 访问没有赋值的自变量

针对单相和多相的模型（比如定义源项，性质和外形），大多数标准的 UDF，你的函数所需要的变量(e.g., domain or thread pointers)在求解过程中通过求解器自动做为自变量直接传递给你的 UDF。例如，如果你的 UDF 定义了特殊边界区域(使用 DEFINE_PROFILE)的外形，在 FLUENT 的边界条件面板中，你的 UDF 就和相和混合物相关联了，在执行时通过求解器，合适的相和混合物变量传递给你的函数。然而，并非所有的 UDF 都直接把函数所需要的自变量传递给求解器。回想一下，例如 DEFINE_ADJUST 和 DEFINE_INIT UDFs 传递给 混合物控制区变量，这里 DEFINE_ON_DEMAND UDFs 是没有被传递的自变量。这部分提供了通过 DEFINE 函数访问没有被直接传递给 UDF 文件的工具。

- 6.5.1 Get_Domain
- 获得控制区
- 6.5.2 Phase Domain Pointer Using the Phase Domain Index (DOMAIN_SUB_DOMAIN)
- 通过相控制区索引号使用相控制区指针
- 6.5.3 Phase-Level Thread Pointer Using the Phase Domain Index (THREAD_SUB_THREAD)
- 通过相控制区索引号使用相等值线指针
- 6.5.4 Phase Thread Pointer Array Using Mixture-Level Thread (THREAD_SUB_THREADS)
- 通过混合物等值线使用相线指针数组
- 6.5.5 Mixture Domain Pointer Using a Phase Domain Pointer (DOMAIN_SUPER_DOMAIN)
- 通过相控制区指针调用混合物控制区指针
- 6.5.6 Mixture Thread Pointer Using a Phase Thread Pointer (THREAD_SUPER_THREAD)
- 通过相线指针使用混合物线指针
- 6.5.7 Thread Pointer Using a Zone ID (Lookup_Thread)
- 通过控制区 ID 使用线指针
- 6.5.8 domain_id Using a Phase Domain Pointer (DOMAIN_ID)
- 通过相控制区指针使用控制区 ID
- 6.5.9 Phase Domain Index Using a Phase Domain Pointer (PHASE_DOMAIN_INDEX)
- 通过相控制区指针使用相控制区索引号

6.5.1 Get_Domain

若控制区指针没有显式地作为自变量传递给你地 UDF，你可以用 Get_Domain 宏恢复控制区指针。

```
Get_Domain(domain_id);
```

domain_id 是一个整数，混合物控制区其值为 1，在多相混合物模型中其值依次加一。注意： Get_Domain(1) 来代替以前的 FLUENT 版本中外部变量 Domain *domain 表达式。（参见 1.4 部分）

Single-Phase Flows

单相流

在单相流中， domain_id 为 1 ， Get_Domain(1) 将放回流体控制区指针。

```
DEFINE_ON_DEMAND(my_udf)
{
  Domain *domain;           /* domain is declared as a variable   */
  domain = Get_Domain(1);   /* returns fluid domain pointer       */
  ...
}
```

Multiphase Flows

多相流

在多相流中，Get_Domain 的返回值或者是混合物等值线，或单相等值线，或相等值线或相等值线控制区指针。domain_id 的值在混合物控制区始终是一，你可以用 FLUENT 里的图形用户界面获得 domain_id。简单的说，在 FLUENT 的相面板中，选择所需的相。domain_id 将被显示出来。你需要用硬件代码整数 ID 作为自变量传递给宏。

```
DEFINE_ON_DEMAND(my_udf)
{
  Domain *mixture_domain;
  mixture_domain = Get_Domain(1);    /* returns mixture domain pointer */
                                      /* and assigns to variable
*/

  Domain *subdomain;
  subdomain = Get_Domain(2); /* returns phase with ID=2 domain pointer*/
                             /* and assigns to variable
*/
  ...
}
```

例子：

下面是一个名为 get_coords 的 UDF 打印出了为两个指定的线 ID 面的线面的质心。这一函数执行在单相流中用的 Get_Domain 工具。在这个例子中，函数 Print_Thread_Face_Centroids 用了 Lookup_Thread 函数来决定线的指针，然后输出了具体文件中线的所有的面的质心。调用 Get_Domain(1) 函数返回控制区的指针（或在多相流中是混合物控制区）。这一变量没有被传递给 DEFINE_ON_DEMAND。

```
/*****************************************************************/
/*  Example of UDF for single phase that uses Get_Domain utility */
/*****************************************************************/

#include "udf.h"

FILE *fout;

Print_Thread_Face_Centroids(Domain *domain, int id)
{
  real FC[2];
  face_t f;
  Thread *t = Lookup_Thread(domain, id);

  fprintf(fout,"thread id %d\n", id);
  begin_f_loop(f,t)
    {
     F_CENTROID(FC,f,t);
     fprintf(fout, "f%d %g %g %g\n", f, FC[0], FC[1], FC[2]);
    }
  end_f_loop(f,t)
  fprintf(fout, "\n");
}

DEFINE_ON_DEMAND(get_coords)
{
  Domain *domain;
  domain = Get_Domain(1);
  fout = fopen("faces.out", "w");
  Print_Thread_Face_Centroids(domain, 2);
  Print_Thread_Face_Centroids(domain, 4);
  fclose(fout);
}
```

6.5.2 Phase Domain Pointer Using the Phase Domain Index （ DOMAIN_SUB_DOMAIN)

6.5.2 通过相控制区索引号使用相控制区指针

有两个方法可以获得混合物控制区具体相（或子区）的指针。你或者可以用宏

DOMAIN_SUB_DOMAIN (在下面描述)或 Get_Domain, 在 6.5.1 部分描述。

DOMAIN_SUB_DOMAIN 有两个自变量：mixture_domain 和 phase_domain_index。

这个函数返回给定 phase_domain_index 的相指针。注意：DOMAIN_SUB_DOMAIN 在执行上和 THREAD_SUB_THREAD 宏相似。 （在 6.5.3. 部分描述）

```
int phase_domain_index = 0;            /* primary phase index is 0 */
Domain *mixture_domain;
Domain *subdomain = DOMAIN_SUB_DOMAIN(mixture_domain,phase_domain_index);
```

mixture_domain 是 mixture-level domain 的指针。

当你用包含控制区自变量(e.g., DEFINE_ADJUST) and 的宏 DEFINE 时，自动通过 FLUENT 的求解器传递给你的 UDF 文件，你的 UDF 文件就和混合物相关联了。否则，如果 mixture_domain 没有显式地传递给你的 UDF，你需要在调用 sub_domain_loop 之前，用另外一个宏工具来恢复(e.g., Get_Domain(1)) 。参见 6.5.1 部分。

phase_domain_index 是子区指针地索引号。 它是一个整数初始相值为 0，以后每相依次加 1。当你用包含相控制区变量(DEFINE_EXCHANGE_PROPERTY, DEFINE_VECTOR_EXCHANGE_PROPERTY)的 DEFINE 宏时，phase_domain_index 是自动通过 FLUENT 的求解器传递给你的 UDF，你的 UDF 就和互相作用的相相联系了。否则，你需要硬代码调用 DOMAIN_SUB_DOMAIN 宏给 phase_domain_index 指针赋值。如果你的多相流模型有两相，然后 phase_domain_index 初始相的值是 0，第二相的值为 1。然而，如果多相流模型中有更多的相，你需要用 PHASE_DOMAIN_INDEX 宏来恢复与给定控制区的 phase_domain_index。详情参考 6.5.9 部分。

6.5.3 Phase-Level Thread Pointer Using the Phase Domain Index （ THREAD_SUB_THREAD)

6.5.3 通过相控制区索引号使用相等值线指针

THREAD_SUB_THREAD 宏可以用来恢复给定相控制区索引号的 phase-level thread (subthread) 指针。 THREAD_SUB_THREAD 有两个自变量： mixture_thread 和 phase_domain_index。这一函数返回给定 phase_domain_index 的 phase-level 线指针。 注意：THREAD_SUB_THREAD 在执行上与 DOMAIN_SUB_DOMAIN 宏相似，参见 6.5.2.部分。

```
int phase_domain_index = 0;               /* primary phase index is 0  */
Thread *mixture_thread;                  /* mixture-level thread pointer */
Thread *subthread = THREAD_SUB_THREAD(mixture_thread,phase_domain_index);
```

mixture_thread 是一 mixture-level 线的指针。当你用包含控制区自变量(e.g., DEFINE_ PROFILE) and 的宏 DEFINE 时，自动通过 FLUENT 的求解器传递给你的 UDF 文件，你的 UDF 文件就和混合物相关联了。否则，如果混合物控制线指针没有显式地传递给你的 UDF，你需要在调用 Lookup_Thread 宏之前，用另外一个宏工具来恢复(e.g., Get_Domain(1)）。参见 6.5.1 部分。 参考 6.5.7 部分)。

phase_domain_index 子区指针的索引号。它是一个整数初始相值为 0，以后每相依次加 1。当你要用包含相控制区索引号变量(DEFINE_EXCHANGE_PROPERTY, DEFINE_VECTOR_EXCHANGE_PROPERTY)的 DEFINE 宏时， phase_domain_index 通过 FLUENT 的求解器自动传递给你的 UDF，你的 UDF 就和具体的相互作用相相互关联了。（参考 UDF 的例子见 4.4.2 部分）否则， 你需要用硬代码改变宏 THREAD_SUB_THREAD 的 phase_domain_index 值。如果你的多相流模型中只有两相，那么 phase_domain_index 对初始相是 0，第二个相为 1。然而，如果你有更多的相，你需要用 PHASE_DOMAIN_INDEX 宏来恢复与给定区域相关的 phase_domain_index 。详情参考 6.5.9 部分。

6.5.4 Phase Thread Pointer Array Using Mixture-Level Thread (THREAD_SUB_THREADS)

6.5.4 通过混合物等值线使用相线指针数组

THREAD_SUB_THREADS 宏 可以用以恢复指针数组, pt, 它的元素包含相等值线（子线）的指针。 THREADS_SUB_THREADS 有一个变量 mixture_thread。

```
Thread *mixture_thread;
```

```
Thread **pt;    /* initialize pt   */
pt = THREAD_SUB_THREADS(mixture_thread);
```

mixture_thread 是 mixture-level thread 代表网格线或面线的指针。当你用包含线变量(e.g., DEFINE_PROFILE)的 DEFINE 宏时，通过 FLUENT 的求解器自动传递给你的 UDF，这个函数就和混合物有关了。否则，如果混合物线的指针没有显式地传递给你的 UDF，然后你需要用令一个方法来恢复。例如：你可以用 Lookup_Thread 宏。(参考 6.5.7 部分)。

pt[i] 数组的元素是与第 I 相的相等值线有关的值，这里 i 是 phase_domain_index。当你想恢复网格具体相的信息时，你可以用 pt[i] 做为一些网格变量宏的自变量。例如： C_R(c,pt[i])可以用来返回网格点 c 第 I 相的密度。 指针 pt[i] 可以用 THREAD_SUB_THREAD 来恢复，在 6.5.3 部分讨论，用 I 做为自变量。phase_domain_index 可以用宏 PHASE_DOMAIN_INDEX 来恢复，参见 6.5.9 部分。

6.5.5 Mixture Domain Pointer Using a Phase Domain Pointer (DOMAIN_SUPER_DOMAIN)

6.5.5 通过相控制区指针调用混合物控制区指针

当你的 UDF 有权访问特殊的相等值线（子区）指针，你可以用宏

DOMAIN_SUPER_DOMAIN,恢复混合物等值线控制区指针。 DOMAIN_SUPER_DOMAIN 含有一个变量 subdomain。注意： DOMAIN_SUPER_DOMAIN 在执行上和 THREAD_SUPER_THREAD 宏是非常相似的。参考 6.5.6.部分。

```
Domain *subdomain;
Domain *mixture_domain = DOMAIN_SUPER_DOMAIN(subdomain);
```

Subdomain 是多相流混合物控制区相等值线的指针。当你用包含控制区变量（e.g., DEFINE_ADJUST）的 DEFINE 宏时，通过 FLUENT 的求解器，它可以自动传递给你的 UDF 文件,这个函数就会和混合物中的第一相和第二相相关了。注意：在当前的 FLUENT 版本中， DOMAIN_SUPER_DOMAIN 将返回与 Get_Domain(1)相同的指针。这样，如果你的 UDF 可以使用子区的指针，建议使用宏 DOMAIN_SUPER_DOMAIN 来代替 Get_Domain 宏以避免将来的 FLUENT 版本造成的不兼容问题。

6.5.6 Mixture Thread Pointer Using a Phase Thread Pointer (THREAD_SUPER_THREAD)

6.5.6 通过相线指针使用混合物线指针

当你的 UDF 有权访问某一条相线（）子线指针你想恢复混合物的等值线指针时，你 可 以 使 用 宏 THREAD_SUPER_THREAD 。 THREAD_SUPER_ 有 一 个 自 变 量 subthread。

```
Thread *subthread;
Thread *mixture_thread = THREAD_SUPER_THREAD(subthread);
```

subthread 在多相流混合物中是一个特殊的相等值线指针。当你使用包含线变量（e.g., DEFINE_PROFILE）的 DEFINE 宏时，通过 FLUENT 的求解器它自动传递给你的 UDF 文件，这个函数就和混合物中的两相相互关联了。注意：在执行上和 THREAD_SUPER_THREAD 宏是非常相似的。参考 6.5.5. 部分。

6.5.7 Thread Pointer Using a Zone ID (Lookup_Thread)

6.5.7 通过区的 ID 使用线指针

当你想要在 FLUENT 的边界条件面板中恢复与给定区域 ID 的线指针时，你可以使用宏 Lookup_Thread 。例如，假设你的 UDF 需要对该区域的特殊线操作（并非查找所有的线），你可以用 DEFINE 宏定义你的 UDF 文件，而不用将线指针(e.g., DEFINE_ADJUST）从 FLUENT 的求解器传递到你的 UDF 文件。你的 UDF 还可以使用 Lookup_Thread 来获得你想要的指针。这一过程分两步：首先，你应该从 FLUENT 的边界条件面板中区域的 ID；然后，你需要使用硬代码做为自变量调用宏 Lookup_Thread。 Lookup_Thread 返回与给定区域 ID 相关的线的指针。你可以将线指针赋给 thread_name，在你的 UDF 中使用。

```
int zone_ID;
Thread *thread_name = Lookup_Thread(domain,zone_ID);
```

在多相流的上下文中，通过宏 Lookup_Thread 返回的线是与控制区自变量相关的相的等值线。

例子

下面是一个使用宏 Lookup_Thread 的 UDF 文件，在这个例子中，通过宏 Lookup_Thread 返回给定区域 ID 的线指针，将它赋给线。 在 begin_f_loop 和 F_CENTROID 中使用这个线指针查找与这个线相关的所有的面，获得面的质心输出到文件。

```
/*******************************************************************/
/*  Example of UDF that uses Lookup_Thread macro                   */
/*  Note If UDF is applied to a multiphase flow problem, the domain*/
/*  passed will be the mixture domain so that the thread that is   */
```

```
/*  returned is the mixture-level thread                          */
/*******************************************************************/

#include "udf.h"

/* domain passed to Adjust function is mixture domain for multiphase*/

DEFINE_ADJUST(print_f_centroids, domain)
{
  real FC[2];
  face_t f;
  int ID = 1;
       /* Zone ID for wall-1 zone from Boundary Conditions panel */
  Thread *thread = Lookup_Thread(domain, ID);

  begin_f_loop(f, thread)

     {
        F_CENTROID(FC, f, thread);
        printf("x-coord = %f  y-coord = %f", FC[0], FC[1]);
     }
  end_f_loop(f, thread)
}
```

6.5.8 domain_id Using a Phase Domain Pointer (DOMAIN_ID)

6.5.8 使用相的控制区指针

当你有权访问与给定相等值线控制区指针的 domain_id 时，你可以使用 DOMAIN_ID。DOMAIN_ID 有一个自变量 subdomain,它是相等值线控制区的指针。控制区（混合物）的最大的等值线的 domain_id 的默认值是 1。即：如过被传递给 DOMAIN_ID 的控制区指针是混合物控制区的等值线指针，那么，函数的返回值为 1。注意：当你在 FLUENT 的相面板中选择需要的相时，它所返回的 domain_id 是和显示在图形用户界面中的整数值 ID 相同的。

```
Domain *subdomain;
int domain_id = DOMAIN_ID(subdomain);
```

6.5.9 Phase Domain Index Using a Phase Domain Pointer (PHASE_DOMAIN_INDEX)

6.5.9 通过相控制区使用相控制区索引号

宏 PHASE_DOMAIN_INDEX 返回给定相等值线控制区（子区）指针的 phase_domain_index。 PHASE_DOMAIN_INDEX 有一个自变量， subdomain,它是 phase-level domain 的指针。 phase_domain_index 是子区指针的索引号。初始相的值为整数 0，以后每相依次加 1。

```
Domain *subdomain;
int phase_domain_index = PHASE_DOMAIN_INDEX(subdomain);
```

6.6 Accessing Neighboring Cell and Thread Variables

6.6 访问邻近网格和线的变量

你可以用 F luent Inc. 提供的宏来确定邻近网格面。在复杂的 UDF 文件中，当你查询特定网格或线的面时，可能会用到这个信息。对给定的面 f 和它的线 tf，两个相邻的网格点为 c0 和 c1。若是控制区附面层上的面则只有 c0，c1 的值为 NULL。一般的，然而 当把网格导入到 FLUENT 中时，按照右手定则定义面上节点的顺序，面 f 上的网格点 c0、c1 都存在。下面的宏返回网格点 c0 和 c1 的 ID 和所在的线。

```
cell_t c0 = F_C0(f,tf);    /* returns ID for c0*/
tc0 = THREAD_T0(tf);       /* returns the cell thread for c0 */
cell_t c1 = F_C1(f,tf);    /*returns ID for c1 */
tc1 = THREAD_T1(tf);       /* returns the cell thread for c1 */
```

回忆由 F_AREA 和 F_FLUX 返回的信息是直接相关的，这些值从网格 c0 到 c1 返回正值。

6.7 User-Defined Memory for Cells (C_UDMI)

6.7 用户为网格定义内存

为了存储、恢复由 UDF 网格区域变量的值，你可以用 C_UDMI 函数分配 500 个单元。这些值可以用做后处理，例如， 通过其它的 UDFs。这个在用户定义内存中存储变量的方法是比用户定义标量(C_UDSI)更有效。

宏: C_UDMI(c, thread, index)

自变量类型： cell_t c

Thread *thread

int index

函数返回值： void

C_UDMI 有三个自变量：c，thread，和 index。 c 是网格标志符号， thread 是网格线指针， index 是识别数据内存分配的。与索引号 0 相关的用户定义的内存区域为 0， （或 udm-0）。

在你用来在内存中存放变量之前，首先你需要在 FLUENT 的 User-Defined Memory 面板中分配内存。参考 8.1.5 部分

Define →User-Defined →Memory...

!!当在分配内存之前，如果你想用 C_UDMI，就会出现错误。

你在图形用户窗口分配的每一个用户定义的内存，都会创建一个新的变量。例如：你要指定两个内存分配区，那么两个变量 udm-0 和 and udm-1 就会在数据储存器中产生。这些名字将会在后台处理面板中显示出来。下面是计算网格点的温度的例子，然后存放到用户定义的内存中。

例子：

```
/* Computes cell temperature and then stores the value in            */
/* user-defined memory location 0 (corresponding to udm-0)           */

begin_c_loop(c, thread)
  {
    temp = C_T(c, thread);
    C_UDMI(c, thread, 0) = (temp - tmin)   / (tmax-tmin);
  }
end_c_loop(c, thread)
```

6.8 矢量工具

Fluent 提供了一些工具，你可以用来在 UDF 中计算有关矢量的量。这些工具在源程序中以宏的形式运行。例如 你可以用实函数 NV_MAG(V)计算矢量 V 的大小(模)。另外你可以用函数 NV_MAG2(V)获得矢量 V 模的平方。下面是在 UDF 中可以利用的矢量工具列表。在矢量工具宏中有个约定俗成的惯例，V 代表矢量，S 代表标量，D 代表一系列三维的矢量，最后一项在二维计算中被忽略。

在矢量函数中约定的计算顺序括号、指数、乘除、加减(PEMDAS)不再适用。相反下划线符号(_) 用来表示一组操作数，因此对元素的操作先于形成一个矢量。

!! 注意：这部分所有的矢量工具都用在 D 和 3D 中。因此，你没有必要在你的 UDF 中做任何的测试。

- 6.8.1 NV_MAG
- 6.8.2 NV_MAG2
- 6.8.3 ND_ND
- 6.8.4 ND_SUM
- 6.8.5 ND_SET
- 6.8.6 NV_V
- 6.8.7 NV_VV
- 6.8.8 NV_V_VS
- 6.8.9 NV_VS_VS
- 6.8.10 ND_DOT

6.8.1 NV_MAG

NV_MAG 计算矢量的大小，即矢量平方和的平方根。

```
NV_MAG(x)

    2D:  sqrt(x[0]*x[0] + x[1]*x[1]);
    3D:  sqrt(x[0]*x[0] + x[1]*x[1] + x[2]*x[2]);
```

6.8.2 NV_MAG2

NV_MAG2 计算矢量的平方和。

NV_MAG2(x)

```
2D:  (x[0]*x[0] + x[1]*x[1]);
3D:  (x[0]*x[0] + x[1]*x[1] + x[2]*x[2]);
```

6.8.3 ND_ND

在 RP_2D (FLUENT 2D) 和 RP_3D (FLUENT 3D)中，常数 ND_ND 定义为 2 。如果你想在 2D 中建立一个 2×2 矩阵，或在 3D 中建立一个 3×3 矩阵，可以用到它。当你用 ND_ND 时,你的 UDF 可以在 2D 和 3D cases 中使用，不用做任何改动。

```
real A[ND_ND][ND_ND]

for (i=0; i<ND_ND; ++i)
  for (j=0; J<ND_ND; ++j)
    A[i][j] = f(i, j);
```

6.8.4 ND_SUM

ND_SUM 计算 ND_ND 的和。

ND_SUM(x, y, z)

```
2D:  x + y;
3D:  x + y + z;
```

6.8.5 ND_SET

ND_SET 产生 ND_ND 任务说明。

ND_SET(u, v, w, C_U(c, t), C_V(c, t), C_W(c, t))

```
u = C_U(c, t);
v = C_V(c, t);
```

if 3D:

```
w = C_W(c, t);
```

6.8.6 NV_V

NV_V 完成对两个矢量的操作。

NV_V(a, =, x);

a[0] = x[0]; a[1] = x[1]; etc.

注意：如果你在上面的方程中用 + = 代替=, 将得到

a[0]+=x[0]; 等。

6.8.7 NV_VV

NV_VV 完成对矢量的基本操作。在下面的宏调用中，这些操作用符号(-,/,*) 代替+。

NV_VV(a, =, x, +, y)

2D: a[0] = x[0] + y[0], a[1] = x[1] + y[1];

6.8.8 NV_V_VS

NV_VS_VS用来把两个矢量相加（后一项都乘一常数）。
NV_V_VS(a, =, x, +, y, *, 0.5);

2D: a[0] = x[0] + (y[0]*0.5), a[1] = x[1] +(y[1]*0.5);

注意：符号+ 可以换成 -, *, 或/, 符号 * 可以替换成 /。

6.8.9 NV_VS_VS

NV_VS_VS 用来把两个矢量相加（每一项都乘一常数）。

NV_VS_VS(a, =, x, *, 2.0, +, y, *, 0.5);

2D: a[0] = (x[0]*2.0) + (y[0]*0.5), a[1] = (x[1]*2.0) + (y[1]*0.5);

注意：符号+ 可以换成 -, *, 或/, 符号 * 可以替换成 /。

6.8.10 ND_DOT

下列的工具计算两个矢量的点积。

ND_DOT(x, y, z, u, v, w)

2D: (x*u + y*v);
3D: (x*u + y*v + z*w);

```
NV_DOT(x, u)

    2D:  (x[0]*u[0] + x[1]*u[1]);
    3D:  (x[0]*u[0] + x[1]*u[1] + x[2]*u[2]);


NVD_DOT(x, u, v, w)

    2D:  (x[0]*u + x[1]*v);
    3D:  (x[0]*u + x[1]*v + x[2]*w);
```

6.9 与非定常数值模拟有关的宏

用 RP 变量宏有权访问 UDF 中非定常的变量。比如：

你的 UDF 可以利用 RP_Get_Real 宏获得流动时间。

```
real current_time;
current_time = RP_Get_Real("flow-time");
```

下面是在每个时间步长处理时间信息的 RP 宏列表。

RP_Get_Real("flow-time")	返回当前的计算时间 (秒)
RP_Get_Real("physical-time-step")	返回当前的计算时间 步长(秒)
RP_Get_Integer("time-step")	返回当前的计算时间 步长数(秒) that have been run

在 函数中记下时间步长的数是有用的，可以用来自判断当前迭代次数是否是第一个时间步长。

下面的例子说明了这个问题。

```
/*********************************************************************
***/
/*  Example UDF that uses an RP Variable Macro
*/
/*********************************************************************
***/
```

```
static int last_ts = -1;   /*  Global variable.  Time step is never <0
*/

DEFINE_ADJUST(first_iter_only, domain)
{
  int curr_ts;
  curr_ts = RP_Get_Integer("time-step");
  if (last_ts != curr_ts)
  {
    last_ts = curr_ts;

    /* things to be done only on first iteration of each time step */
    /* can be put here  */

  }
}
```

6.10 其它各种工具宏

- 6.10.1 Data_Valid_P
- 6.10.2 FLUID_THREAD_P
- 6.10.3 NULLP & NNULLP
- 6.10.4 BOUNDARY_FACE_THREAD_P(t)
- 6.10.5 C_FACE_THREAD(c,t,i)
- 6.10.6 C_FACE(c,t,i)
- 6.10.7 M_PI
- 6.10.8 UNIVERSAL_GAS_CONSTANT
- 气体常数宏
- 6.10.9 SQR(k) and SQRT(k)
- 6.10.10 Message Macro
- 信息宏
- 6.10.11 Error Macro
- 错误宏

6.10.1 Data_Valid_P

在你用 Data_Valid_P 宏 进行计算之前，你可以检查函数中出现的网格单元变量值。

```
boolean Data_Valid_P()
```

Data_Valid_P 定义在 id.h 头文件中， 包含在 udf.h 中。数据有效返回 1（true），否则返回 0（false）。

例子：

```
if(!Data_Valid_P())                    return;
```

比如：假设你要读进一个 cas 文件，要导入 UDF。如果你要用没有初始化的变量（比如内场单元的速度）来完成计算，将会出错。为避免这类错误，可以在你的源代码中加入 if else 条件。若满足 if 的条件，函数按照正常执行，若不满足则什么也不做。流场一初始化，就激活函数进行正确的计算。

6.10.2 FLUID_THREAD_P

你可以用 FLUID_THREAD_P 函数来检查一个线是否是流体中的线。如果你定义的线有效，这个函数返回 1（true)，否则返回 0（false）。

```
boolean FLUID_THREAD_P()
```

例子：

```
FLUID_THREAD_P(t0)
```

6.10.3 NULLP & NNULLP

你可以用 NULLP 和 NNULLP 函数来检查用户定义的量是否被分配了内存。

如果没有分配内存的话，NULLP 返回 TRUE；如果分配内存的话，NNULLP 返回 TRUE。下面是这个用法的例子：

```
NULLP(T_STORAGE_R_NV(t0, SV_UDSI_G(p1)))

/*  NULLP returns TRUE if storage is not allocated for  */
/*  user-defined storage variable                       */
NNULLP(T_STORAGE_R_NV(t0, SV_UDSI_G(p1)))

/*  NNULLP returns TRUE if storage is allocated for     */
/*  user-defined storage variable                       */
```

6.10.4 BOUNDARY_FACE_THREAD_P(t)

如果 Thread *t 是边界面线，这个宏返回 TRUE 。

6.10.5 C_FACE_THREAD(c,t,i)

这个宏返回面 face_t f（由 C_FACE 返回）的 Thread *t。注意：索引号 i 可以被传递给 c_face_loop。

6.10.6 C_FACE(c,t,i)

这个宏返回给定单元 cell_t c 和 Thread *t 的面 face_t f。注意：索引号 i 可以被传递给 c_face_loop。

6.10.7 M_PI

M_PI 宏返回给定值 π。

6.10.8 UNIVERSAL_GAS_CONSTANT

UNIVERSAL_GAS_CONSTANT 宏 返回气体常数的值（8314.34），单位为国际单位 *J/ Kmol- K*。

6.10.9 SQR(k) and SQRT(k)

SQR(k) 返回给定变量 k 的平方，*k * k*。

SQRT(k) 返回给定变量 k 的平方根 $\sqrt{k}$.

6.10.10 信息宏

当你想要把变量输出到控制窗口时，你可以用信息宏。在下面的例子中，紊流扩散的体积分：将被显示在控制窗口中，变量的迭代值 sum_diss 将在所有的信息中被取代比如用%g。建议你在编译 UDFs (UNIX only)时，用 Message 代替 printf。

```
Message ("Volume integral of turbulent dissipation: %g\n", sum_diss);
```

6.10.11 错误宏

当你想要停止 UDF 的执行或者把错误信息打印到控制窗口时，你可以用错误宏。

```
if (table_file == NULL)
```

```
Error("error reading file");
```

!! 这个宏仅仅在编 UDF 文件时有效。

第七章　UDF的编译与链接

编写好UDF件（详见第三章）后，接下来则准备编译（或链接）它。在7.2或7.3节中指导将用户编写好的UDF如何解释、编译成为共享目标库的UDF。

_ *第 7.1 节: 介绍*

_ *第 7.2 节: 解释 UDF*

_ *第 7.3 节: 编译 UDF*

7.1　介绍

解释的UDF和编译的UDF其源码产生途径及编译过程产生的结果代码是不同的。编译后的UDF由C语言系统的编译器编译成本地目标码。这一过程须在FLUENT运行前完成。在FLUENT运行时会执行存放于共享库里的目标码，这一过程称为“动态装载”。

另一方面，解释的UDF被编译成与体系结构无关的中间代码或伪码。这一代码调用时是在内部模拟器或解释器上运行。与体系结构无关的代码牺牲了程序性能，但其UDF可易于共享在不同的结构体系之间，即操作系统和FLUENT版本中。如果执行速度是所关心的，UDF文件可以不用修改直接在编译模式里运行。

为了区别这种不同，在FLUENT中解释UDF和编译UDF的控制面板其形式是不同的。解释UDF的控制面板里有个“Compile按钮”，当点击“Compile按钮”时会实时编译源码。编译UDF的控制面板里有个“Open按钮”，当点击“Open按钮” 时会“打开”或连接目标代码库运行

FLUENT（此时在运行FLUENT之前需要编译好目标码）。

当FLUENT程序运行中链接一个已编译好的UDF库时，和该共享库相关的东西都被存放到case文件中。因此，只要读取case文件，这个库会自动地链接到FLUENT处理过程。同样地，一个已经经过解释的UDF文件在运行时刻被编译，用户自定义的C函数的名称与内容将会被存放到用户的case文件中。只要读取这个case文件，这些函数会被自动编译。

注：已编译的UDF所用到的目标代码库必须适用于当前所使用的计算机体系结构、操作系统以及FLUENT软件的可执行版本。一旦用户的FLUENT升级、操作系统改变了或者运行在不同的类型的计算机，必须重新编译这些库。

UDF必须用DEFINE宏进行定义，DEFINE宏的定义是在udf.h文件中。因此，在用户编译UDF之前，udf.h文件必须被放到一个可被找到的路径，或者放到当前的工作目录中。

udf.h文件放置在：

path/Fluent.Inc/fluent6.+*x*/src/udf.h

其中path是Fluent软件的安装目录，即Fluent.Inc目录。X代表了你所安装的版本号。

通常情况下，用户不应该从安装默认目录中复制udf.h文件。编译器先在当前目录中寻找该文件，如果没找到，编译器会自动到/src目录下寻找。如果你升级了软件的版本，但是没有从你的工作目录中删除旧版本的udf.h文件，你则不能访问到该文件的最新版本。在任何情

况下都不应该改变udf.h文件。

7.2 UDF解释

这一节介绍编译经过解释的UDF的步骤。一旦经过解释的UDF被编译，用户自定义的C函数的名称与内容将会被存放到case文件中。只要读取这个case文件，这些函数便会自动被编译。

编译被解释的UDF的一般程序如下：

1. 如果用户没有在网络Windows计算机上使用并行的FLUENT版本，则需要确定UDF的C源码和case文件与当前工作目录一致。具体步骤见7.2.2节。

！如果源码不在当前工作目录，则用户编译UDF时，用户必须在解释UDF的控制面板里输入文件的完全路径，而不是只输入文件名。

2. 在当前工作目录下运行FLUENT。
3. 读取（或建立）case文件。
4. 打开“Interpreted UDFs panel”，编译UDF（如vprofile.c）。

Define → User-Defined → Functions → Interpreted...

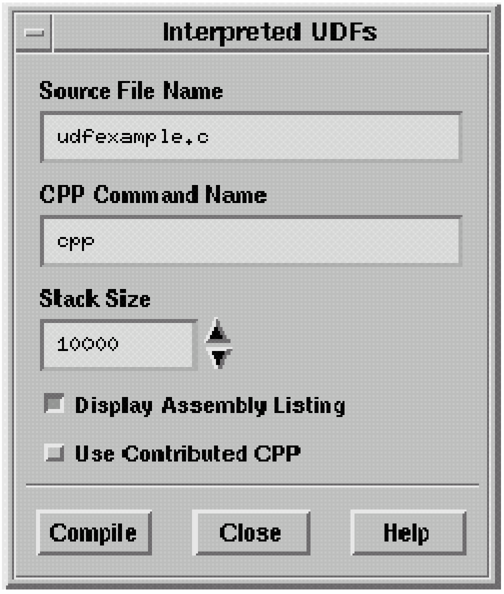


图7.2.1 解释的UDF的控制面板

(a) 在“Source File Name”下输入C源码的文件名（如vprofile.c）。

! 如果自定义的C源码不在工作目录中，用户必须输入完全的自定义的C函数路径。

当写入case文件时，自定义源码的名称（或源码的完全路径）会存放到case文件中。

(b) 在“CPP Command Name”指定为C的预处理程序。当然也有其它有效的ANSI C预处理程序，包括gcc -E和cc -E。详细检查所用的计算机系统管理。

(c) 如果自定义函数局部变量数不引起栈的溢出，则保持“Stack Size”的默认值为1000。此时，所设 “Stack Size” 的数要远远大于局部变量用的数。

(d) 选择“Display Assembly Listing”，则当函数编译汇编码的清

单会出现在窗口的控制台内。这一选项会保存于case文件，当用户接着运行FLUENT时汇编码会自动显示。

(e) 如果用户使用“Fluent Inc”提供的C预处理程序，选择“Use Contributed CPP”。

(f) 点击“Compile”编译UDF。

自定义C程序的名称和内容会存入于所写的case文件。只要编译UDF，汇编码会出现在控制窗口，所示范例如下。

```
inlet_x_velocity:
                .local.pointer thread (r0)
                .local.int nv (r1)
    0           .local.end
    0           save
                .local.int f (r3)
    1           push.int 0
                .local.pointer x (r4)
    3           begin.data 8 bytes, 0 bytes initialized:
    7           save
    .              .
    .              .
    .              .
  156           pre.inc.int f (r3)
  158           pop.int
  159           b .L3 (22)
      .L2:
  161           restore
  162           restore
  163           ret.v
```

(g) 编译结束点击“Close”。

！如果在一次模拟中使用多于一个的UDF，用户需要将这些函数连接在一个C文件中，例如all.c。然后用“Interpreted UDFs”面板编译连接的文件。这些函数可以作为边界条件、源项及特性等。

7.2.2 基于Windows并行网络的目录结构

在基于Windows网络上使用并行FLUENT版本需要专门的方法组织用户文件。具体步骤如下：

1. 在“Fluent.Inc”目录下创建一个名为“udf” 的可写子目录。
2. 在udf目录下创建子目录（如Fluent.Inc\udf\myudf），将UDF的C源码存放于这个目录下。如果在同一串下多个用户运行工作，每个用户在udf目录下创建自己的子目录（如Fluent.Inc\udf\abcudf和xyzudf）。

！因为源码不在当前工作目录下，所以在编译UDF时必须在“Interpreted UDFs”面板中输入文件的完全路径。例如，编译example.c文件时，输入如下：

\\<fileserver>\Fluent.Inc\udf\myudf\example.c

<fileserver>应输入用户所安装FLUENT的计算机名（如myserver）。

3. 确定所建立的case文件在当前工作目录下。

7.2.3 调试解释的UDF

编译UDF时出错信息会出现在控制窗口中。用户有可能因错误滚动太快不能看到所用的出错信息。因此调试UDF时用户想关掉“Display Assembly Listing”。

如果在调试UDF的过程中一直打开“Interpreted UDFs”面板，由于在独立窗口进行编辑，编译按钮则会不断重复编译。然后，直到无出错信息调试和编译才会结束。

下面介绍一个出错例子，即在“Interpreted UDFs”控制面板中，编

译被解释过的UDF时指定了错误的源文件。上面曾介绍过如果仅仅从当前工作目录下启动FLUENT，在“Interpreted UDFs”控制面板中键入用户的C源码的文件名，则case文件和C源码被指定于当前工作目录下。如果用户编译的C源码与工作目录是不同的路径，用户必须输入C源码所在的完整路径。否则会出现以下的错误信息：

```
gcc: vprofile.c: No such file or directory
gcc: No input files
Error: vprofile.c: line 1: syntax error.
```

如果编译UDF写完case文件后，接着移动C源码到不同位置，会在接着运行FLUENT的过程中产生同样的错误信息。

为了避免错误，只需要在“Interpreted UDFs”控制面板中的“Source File Name”下输入完全的路径名，然后点击“Compile”。此时写case文件会保存C源码的新路径。

7.3 编译UDF

这一节介绍如何链接编译好的UDF。这一过程需要使用C编译器。大部分UNIX的操作系统提供了C编译器。如果在PC机上运行，需要安装VC++编译器（如微软C++、v6.0或更高的版本）。一旦编译好的UDF库文件在FLUENT运行时链接到FLUENT处理过程，和共享库相关的东西会保存到case文件。因此，只要读取case文件，编译的库文件会自动链接到FLUENT处理过程。在控制窗口将会出现链接状态的报告如下：

```
Opening library "libp1/ultra/2d/libudf.so"...
    p1_adjust
```

```
        energy_source
        p1_source
        p1_diffusivity
        p1_bc
    Done.
```

7.3.1 一般程序

编译和链接一个编译好的UDF的一般程序如下所示：

1. 在当前工作目录下，建立专门的目录结构（见7.3.2节）。
2. 编译用户的UDF和修建共享库（见7.3.3节）。
3. 在当前工作目录下运行FLUENT。
4. 读取（或建立）case文件（确信case文件在当前工作目录下）。
5. 链接共享库到FLUENT（见7.3.4节）。

7.3.2 建立目录结构

对于UNIX系统和Windows系统来说，目录结构是不同的。下面分别介绍在两种系统下如何建立目录结构。

UNIX系统

对于UNIX系统下编译的UDF来说，makefile.udf和makefile.udf2两个文件在编译UDF库被需要。makefile文件包含了用户自定义部分，在这部分允许输入用户源函数和FLUENT的安装路径。这些文件的完整路

径如下：

path/Fluent.Inc/fluent6.+*x*/src/makefile.udf

path/Fluent.Inc/fluent6.+*x*/src/makefile.udf2

其中*path是用户直接安装*Fluent.Inc的路径，*x是用户安装*Fluent*版本的相应数（如，*fluent6.0为0）。

! FLUENT安装后所释放的makefile.udf2文件名为Makefile.udf。

下面介绍建立共享库所要求的目录结构。通过下面的例子来介绍目录结构的建立，如图7.3.1所示。

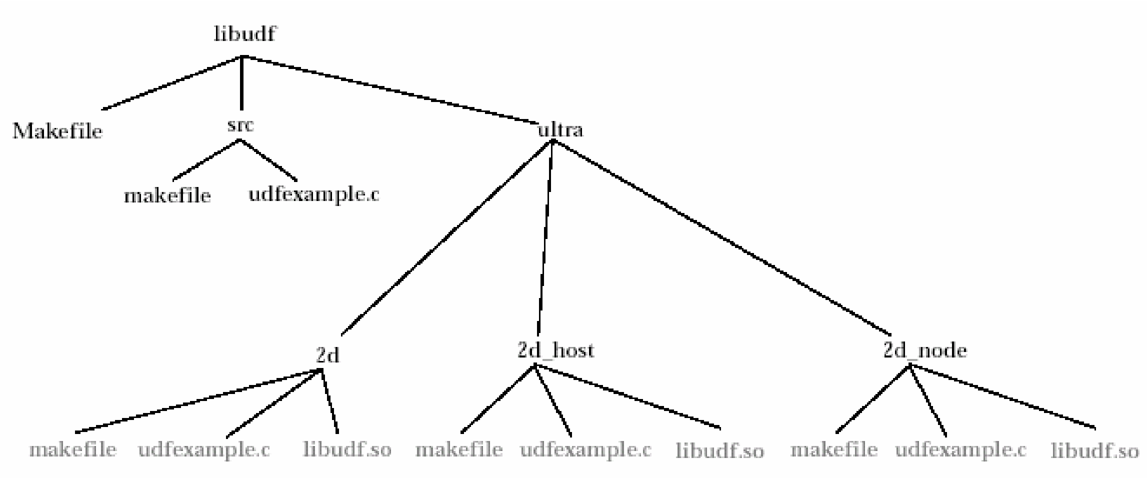


图7.3.1　为编译好的UDF建立库目录的样本（UNIX）

需要注意的是在图7.3.1所示的目录结构为FLUENT的两种版本：二维单精度串型和二维单精度平行。

!不要在目录（2d，2d host，等等）下存放任何文件。当编译用户库（见7.3.3节）时，图7.3.1中所示的文件会自动存放。

1. 在当然工作目录下，创建一个储存用户库的目录（如libudf）。

2. 从以上所示目录下复制makefile.udf2到用户目录（如libudf），

且改名为Makefile。

3. 在用户创建的库目录下，建立一个储存用户源码、命名为src的源码目录。

4. 复制用户源码（如udfexample.c）到所建的/src目录下。

5. 从以上所示目录下复制makefile.udf到用户的/src目录，并且命名为makefile。

6. 鉴别FLUENT所运行的计算机的体系机构。

(a) 开始FLUENT程序。

(b) 向上滚动FLUENT的控制窗口到“\Starting”信息处，鉴定FLUENT的体系机构。

(c) 退出FLUENT程序。

！如果体系机构是irix6.5，需要在makefile进行额外的修改。

7. 为体系机构所建不同版本下创建目录（如ultra/2d和ultra/3d）。存在的版本如下所示：

- single-precision serial 2D or 3D: 2d or 3d
- double-precision serial 2D or 3D: 2ddp or 3ddp
- single-precision parallel 2D or 3D: 2d_node and 2d_host or 3d_node and 3d_host
- double-precision parallel 2D or 3D: 2ddp_node and 2ddp_host or 3ddp_node and 3ddp_host

！需要注意：不管计算节点的数量，用户必须为每个并行版本的求解器（如在三维下有两个目录，二维双精度版本下有两个目录，等等）

创建两个调试目录。

!不要在目录（2d，2d host，等等）下存放任何文件。当编译用户库（见7.3.3节）时，图7.3.1中所示的文件会自动存放。

Windows系统

对于Windows系统下编译的UDF来说，makefile_nt.udf和user_nt.udf两个文件在编译UDF库被需要。user_nt.udf文件中包含了用户自定义部分，在这部分允许输入用户源函数及其它信息。

为了建立共享库所要求的目录结构，需要按照上面所讲的操作步骤进行。通过下面的例子来介绍目录结构的建立，如图7.3.2所示。

需要注意的是在图7.3.2所示的目录结构为FLUENT的两种版本：二维单精度串型和二维单精度并行。

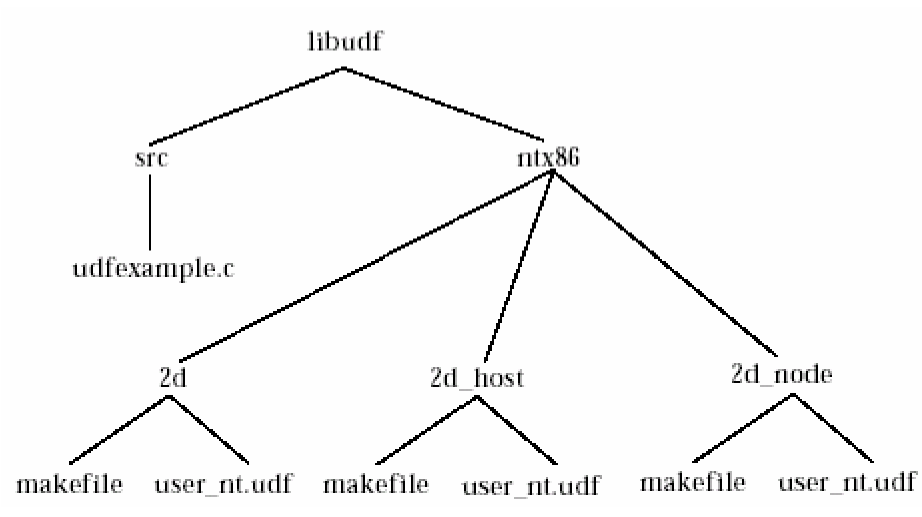


图7.3.2　为编译好的UDF建立库目录的样本（Windows）

1. 在当前工作目录下，创建一个储存用户库的目录（如libudf）。

2. 在用户创建的库目录下，建立一个储存用户源码、命名为src的源

码目录。

3. 复制用户源码（如udfexample.c）到所建的/src目录下。

4. 建立所使用体系机构的目录，如基于Windows的Intel系统使用的目录为ntx86。

5. 针对所用的体系机构建立不同版本的目录（如ntx86\2d）。存在的版本如下所示：

- single-precision serial 2D or 3D: 2d or 3d
- double-precision serial 2D or 3D: 2ddp or 3ddp
- single-precision parallel 2D or 3D: 2d_node and 2d_host or 3d_node and 3d_host
- double-precision parallel 2D or 3D: 2ddp_node and 2ddp_host or 3ddp_node and 3ddp_host

！需要注意：不管计算节点的数量，用户必须为每个并行版本的求解器（如在三维下有两个目录，二维双精度版本下有两个目录，等等）创建两个调试目录。

6.复制makefile_nt.udf和user_nt.udf到相应版本所建的目录下，如2d。

！对于并行的版本来说，一定要复制这两个文件到主机和节点目录，即如图7.3.2所示的2d_node and 2d_host目录。

这两个文件的完整路径如下：

path\Fluent.Inc\fluent6.+*x*\src\makefile_nt.udf

path\Fluent.Inc\fluent6.+*x*\src\user_nt.udf

其中*path是用户直接安装*Fluent.Inc的路径，*x是用户安装*Fluent*版本的相应数（如，*fluent6.0为0）。

！确定makefile_nt.udf和user_nt.udf为最新版本所用文件。如果安装新的FLUENT 6版本，必须复制新的makefile_nt.udf和user_nt.udf到相应的创建目录。

7. 重命名makefile_nt.udf为makefile。

7.3.3 编译和创建用户共享库

下面分别介绍UNIX和Windows系统下如何编译和创建共享库。

UNIX系统

在建立目录并存放文件到相应位置后，便能开始编译和创建共享库。

1. 在用户的src目录下编辑文件makefile，设置参数如下：

- SOURCES = 编译好的用户自定义函数
- FLUENT INC = 用户的安装路径

下面是一个makefile的例子：

```
#-------------------------------------------------------------------------#
# makefile for user defined functions.
#
#-------------------------------------------------------------------------#
```

```
#-------------------------------------------------------
------#
# User modifiable section.
#-------------------------------------------------------
------#
SOURCES= udfexample.c
FLUENT_INC= /path/Fluent.Inc
#-------------------------------------------------------
------#
# Build targets (do not modify below this line).
#-------------------------------------------------------
------#
.
.
```

2. 如果体系机构是irix6.5，还需要在makefile中进行附加变化。

(a) 找到makefile文件中找到下面的命令行

CFLAGS_IRIX6R10= -KPIC -ansi -fullwarn -O -n32

(b) 改变“-ansi”为“-xansi”，即

CFLAGS_IRIX6R10= -KPIC -xansi -fullwarn -O -n32

对于其它的体系机构不需要进行以上变动。

3. 在工作目录（如libudf）下，执行make命令，包含前一步（在7.3.2节）确定的体系机构（如ultra），即

make "FLUENT ARCH=ultra"

控制窗口显示信息为：

```
# linking to ../../src/udfexample.c in ultra/2d
# linking to ../../src/makefile in ultra/2d
# building library in ultra/2d
# linking to ../../src/udfexample.c in ultra/3d
# linking to ../../src/makefile in ultra/3d
# building library in ultra/3d
```

以上面的makefile为例，用户自定义函数udfexample.c被编译和存放于版本所指定的共享库libudf.so中，如在图7.3.1中所示的2d，2d_host和2d_node。虽然在这个例子中只用了一个C函数，但是用户在“SOURCES = in the makefile”下可通过空格分隔多个源文件。

Windows系统

在建立目录并存放文件到相应位置后，便能开始编译和创建共享库。

1. 用文本文件编译user_nt.udf文件，设置参数为：

- SOURCES =编译好的用户自定义函数。在每个文件前加前缀$(SRC)，如 一个函数为$(SRC)udfexample.c，两个函数为$(SRC)udfexample.c $ (SRC)udfexample2.c）。
- VERSION =所用版本，即如图7.3.2所示2d，3d，2ddp，3ddp，2d host，2d node，3d host，3d node，2ddp host，2ddp node，3ddp host，或3ddp node。

● PARALLEL NODE =并行连接库。对于求解器的一系列版本无指定的并行连接库。可能的输入为：

—{ none: serial

—{ smpi: parallel using shared memory (for multiprocessor machines)

—{ vmpi: parallel using shared memory or network with vendor MPI software

—{ net: parallel using network communicator with RSHD software

！如果使用的是并行求解器，需要复制两个user_nt.udf文件（一个是主目录的，一个是节点目录），并且指定两个文件适当的SOURCE, VERSION, and PARALLEL NODE。

下面给出user_nt.udf文件的一个例子：

```
# Replace text in " " (and remove quotes)
#  | indicates a choice
#  note: $(SRC) is defined in the makefile

SOURCES = $(SRC)udfexample.c
VERSION = 2d
PARALLEL_NODE = none
```

对于指定多个用户定义函数只需要在SOURCES下输入它们，并用空格分开即可。

2. 在MS-DOS 命令提示窗口，键入nmake建立目录，如

\libudf\ntx86\2d\。

需要注意的是如果创建存在问题，通过键入nmake clean来进行完整的重建。

7.3.4 连接共享库到FLUENT可执行文件

按照7.3.3小节所示的unix或windows环境下的步骤编译共享库后，就可以将其连接到FLUENT程序中。将共享库连接到FLUENT程序中的步骤如下：

1. 从当前工作目录下启动FLUENT。
2. 读取（或设置）case文件。（确认此case文件保存在当前的工作目录下。）
3. 连接共享库到FLUENT执行文件中。

Define → User-Defined → Functions → Compiled...

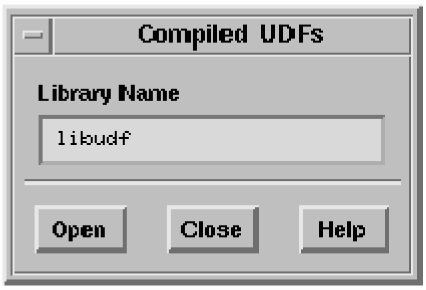


Figure 7.3.3: The Compiled UDFs Panel

(a) 如果共享库保存在当前工作目录中，在Library Name对话框中输入相对路径(例如，libudf)，否则输入完整路径，(例如，~myhome/myfiles/libudf).

！注意，如果存放共享库的目录不在当前的工作目录或其子目录下，

一定要输入完整的路径名。

(b) 单击 Open 按钮，这会将共享库连接到 FLUENT 执行文件中。求解器会自动搜索相应的结构格式和版本。一旦共享库连接到 FLUENT 执行文件以及 case 文件被写入以后，此连接关系就被保存在 case 文件中。因此，无论何时将 case 文件读入 FLURENT，此共享库都会被自动连接。

7.3.5 编译和连接共享库时常见的错误

指定库的名称

仅当从当前的工作目录下运行FLUENT并且共享库的目录是当前工作目录的子目录时，才可以在Compiled UDFs 面板上直接输入共享库的目录(例如libudf)。如果被使用的共享库不在上述位置上，连接此共享库时，必须提供其完整的路径。否则会出现以下错误：

```
 Opening library "libudf/ultra/3d/libudf.so"...
 Error: open_udf_library: couldn't open library:
libudf/ultra/3d/libudf.so
```

如果将共享库移至其他的位置，而要读入的case文件又包含此共享库的连接，也会出现上述错误信息。

解决方法

为了解决这个问题，可以在Library Name in the Compiled UDFs 面

板上输入共享库的完整路径，然后单击Open按钮。虽然同样的错误信息仍将出现，但是新的路径会保存在case文件中。重新读入case文件就会得到正确的连接。

使用不同版本的FLUENT

如果采用一个版本的FLUENT(例如6.0.1)编译UDF，而试图采用另一个不同的版本(例如6.0.2)读入case文件时，会出现以下错误：

```
Error: open_udf_library: library version 6.0.1 incompatible with solver version 6.0.2
```

解决办法

用新版本的FLUENT重新编译UDF，然后再次读入case文件。

第八章　在 FLUENT 中激活你的 UDF

一旦你已经编译（并连接）了你的 UDF，如第 7 章所述，你已经为在你的 FLUENT 模型中使用它做好了准备。根据你所使用的 UDF，遵照以下各节中的指导。

- 8.1 节　激活通用求解器 UDF
- 8.2 节　激活模型明确 UDF
- 8.3 节　激活多相 UDF
- 8.4 节　激活 DPM UDF

8.1　激活通用求解器 UDF

本节包括激活使用 4.2 节中宏的 UDF 的方法。

8.1.1　已计算值的调整

一旦你已经使用 7.2 节和 7.3 节中概括的方法之一编译（并连接）了调整已计算值 UDF，这一 UDF 在 **FLUENT** 中将成为可见的和可选择的。你将需要在 **User-Defined Function Hooks** 面板的 **Adjust Function** 下拉菜单（图 8.1.1）中选择它。

Define → User-Defined → Function Hooks...

调整函数（以 **DEFINE_ADJUST** 宏定义）在速度、压力及其它数量求解开始之前的一次迭代开始的时候调用。例如，它可以用于在一个区域内积分一个标量值，并根据这一结果调整边界条件。有关 **DEFINE_ADJUST** 宏的更多内容将 4.2.1 节。**调整函数**在什么地方适合求解器求解过程方面的信息见 3.3 节。

8.1.2 求解初始化

一旦你已经使用 7.2 节和 7.3 节中概括的方法之一编译（并连接）了求解初始化 UDF，这一 UDF 在 FLUENT 中将成为可见的和可选择的。你将需要在 **User-Defined Function Hooks** 面板的 **Initialization Function** 下拉菜单（图 8.1.1）中选择它。

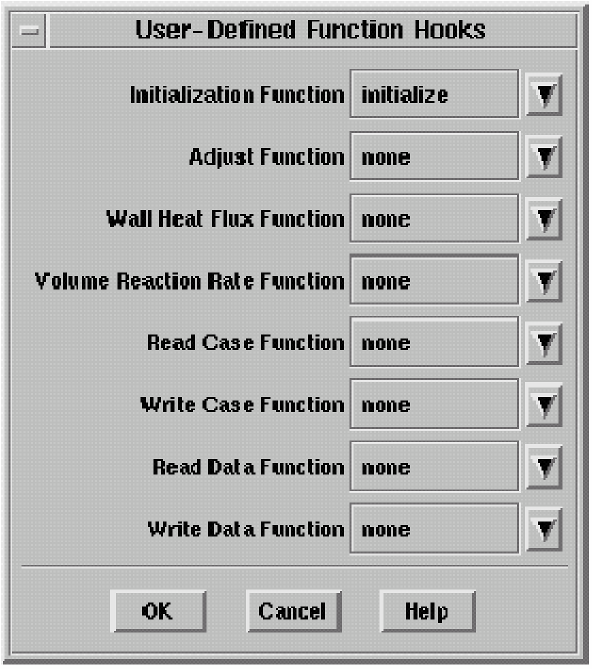


Figure 8.1.1: The User-Defined Function Hooks Panel

Define ⟶ User-Defined ⟶ Function Hooks...

求解初始化 UDF 使用 DEFINE_INIT 宏定义。细节见 4.2.2 节。

8.1.3　用命令执行 UDF

一旦你已经使用 7.2 节和 7.3 节中概括的方法之一编译（并连接）了你的 UDF，你可以在 Execute UDF On Demand 面板中选择它（图 8.1.2），以在某个特定的时间执行这个 UDF，而不是让 FLUENT 在整个计算中执行它。

Define ⟶ User-Defined ⟶ Execute On Demand...



Figure 8.1.2: The Execute On Demand Panel

点击 Execute 按纽让 FLUENT 立即执行它。

以命令执行的 UDF 用 DEFINE_ON_COMMAND 宏定义，更多细节见 4.2.3 节

8.1.4　从 case 和 data 文件中读出及写入

一旦你已经使用 7.2 节和 7.3 节中概括的方法之一编译（并连接）了一个将定制片段从 case 和 data 文件中读出或写入的 UDF,这一 UDF 在 **FLUENT** 中将成为可见的和可选择的。你将需要在 **User-Defined Function Hooks** 面板（图 8.1.1）中选择它。

Define → User-Defined → Function Hooks...

读 Case 函数在你将一个 case 文件读入 **FLUENT** 时调用。它将指定从 case 文件读出的定制片段。

写 Case 函数在你从 **FLUENT** 写入一个 case 文件时调用。它将指定写入 case 文件的定制片段。

读 Data 函数在你将一个 data 文件读入 **FLUENT** 时调用。它将指定从 data 文件读出的定制片段。

写 Data 函数在你从 **FLUENT** 写入一个 data 文件时调用。它将指定写入 data 文件的定制片段。

上述 4 个函数用 **DEFINE_RW_FUCTION** 宏定义，见 4.2.4 节。

8.1.5　用户定义内存

你可以使用你的 UDF 将计算出的值存入内存，以便你以后能重新得到它，要么通过一个 UDF 或是在 **FLUENT** 中用于后处理。为了能访问这些内存，你需要指定在用户定义内存（**User-Defined Memory**）面板中指定用户定义内存单元数量（**Number of User_Defined Memory Locations**）（图 8.1.3）。

Define → User-Defined → Memory...

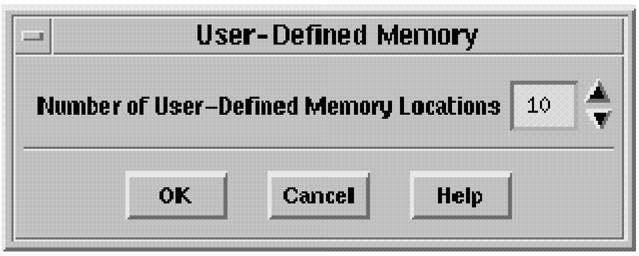


Figure 8.1.3: The User-Defined Memory Panel

宏 **C_UDMI** 或 **F_UDMI** 可以分别用于在你的 UDF 中访问一个单元或面中的用户定义内存位置。细节见 5.2.4，5.3.2，6.7 节。

已经存储在用户定义内存中的场值将在你下次写入一个时存入 data 文件。这些场同样也出现在 FLUENT 后处理面板中下拉列表的 **User Defined Memory…**中。它们将被命名为 udm-0，udm-1 等，基于内存位置索引。内存位置的整个数量限制在 500。

8.2　激活模型明确 UDF

本节包括激活使用 4.3 节中宏的 UDF 的方法。

8.2.1　边界条件

一旦你已经使用 7.2 节和 7.3 节中概括的方法之一编译（并连接）了求解边界条件 UDF，这一 UDF 随之在 FLUENT 中将成为可见的和可选择的，你可以在适当的边界条件面板中选择它。例如，你的 UDF 定义了一个速度入口边界条件，然后你将在 **Veloctiy Inlet** 面板里适当的下拉列表中选择你的 UDF 名字（在你的 C 函数中已经定义，如 inlet_x_velocity）。

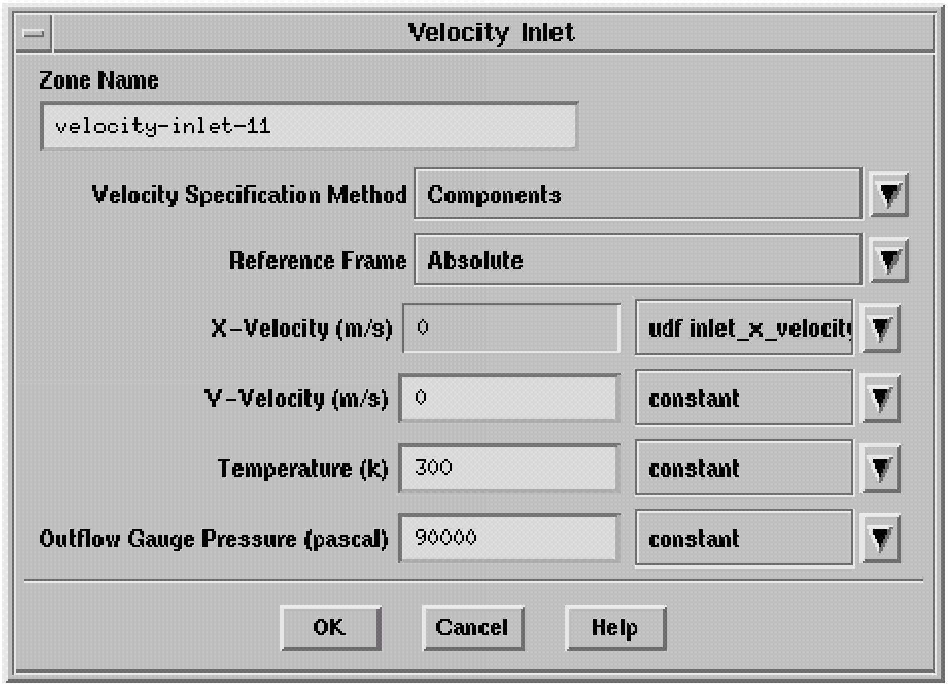


Figure 8.2.1: The Velocity Inlet Panel

如果你使用你的 UDF 指定一个单元区域中的一个固定值，你将需要打开 **Fixed Values** 选项，并在 **Fluid** 或 **Solid** 面板的适当下拉列表中选择你的 UDF 的名字。

边界条件 UDF 用 **DEFINE_PROFILE** 宏定义。细节见 4.3.5 节。

8.2.2 热流量

一旦你已经使用 7.2 节和 7.3 节中概括的方法之一编译（并连接）了求解边界条件 UDF，这一 UDF 随之在 **FLUENT** 中将成为可见的和可选择的，你将需要在 **User-Defined Function Hooks** 面板的 Wall Heat Flux Function 下拉列表（图 8.1.1）中选择它。

Define ⟶ User-Defined ⟶ Function Hooks...

热流量 UDF 用 **DEFINE_HEAT_FLUX** 宏定义。细节见 4.3.3 节。

8.2.3 Nox 产生速率

一旦你已经使用 7.2 节和 7.3 节中概括的方法之一编译（并连接）了 Nox 产生速率 UDF，这一 UDF 随之在 **FLUENT** 中将成为可见的和可选择的，你将需要在 NOx Model 面板中 **User_Defined Functions** 下的 NOx Rate 下拉列表中选择它，如下所示（图 8.2.2）。

Define ⟶ Models ⟶ Pollutants ⟶ NOx...

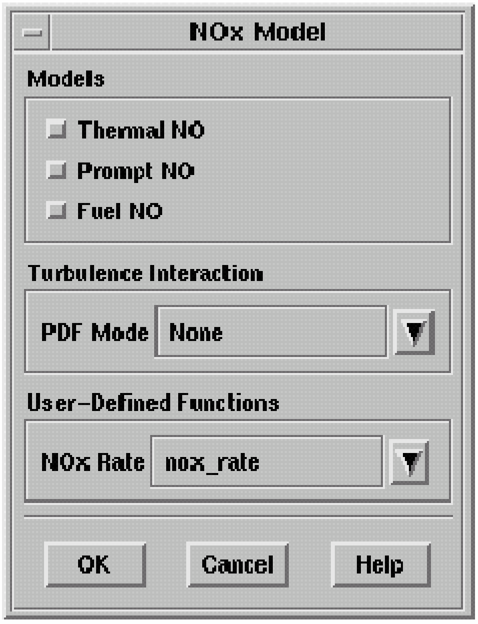


Figure 8.2.2: The NOx Model Panel

Nox 产生速率 UDF 用 **DEFINE_NOX_RATE** 宏定义。细节见 4.3.4 节。

8.2.4　材料属性

一旦你已经使用 7.2 节和 7.3 节中概括的方法之一编译（并连接）了属性定义 UDF，这一 UDF 随之在 FLUENT 中将成为可见的和可选择的，你将首先在 **Materials** 面板中 适当属性的下拉列表中选择 **user-defined**（图 8.2.3）。

Define ⟶ Materials...

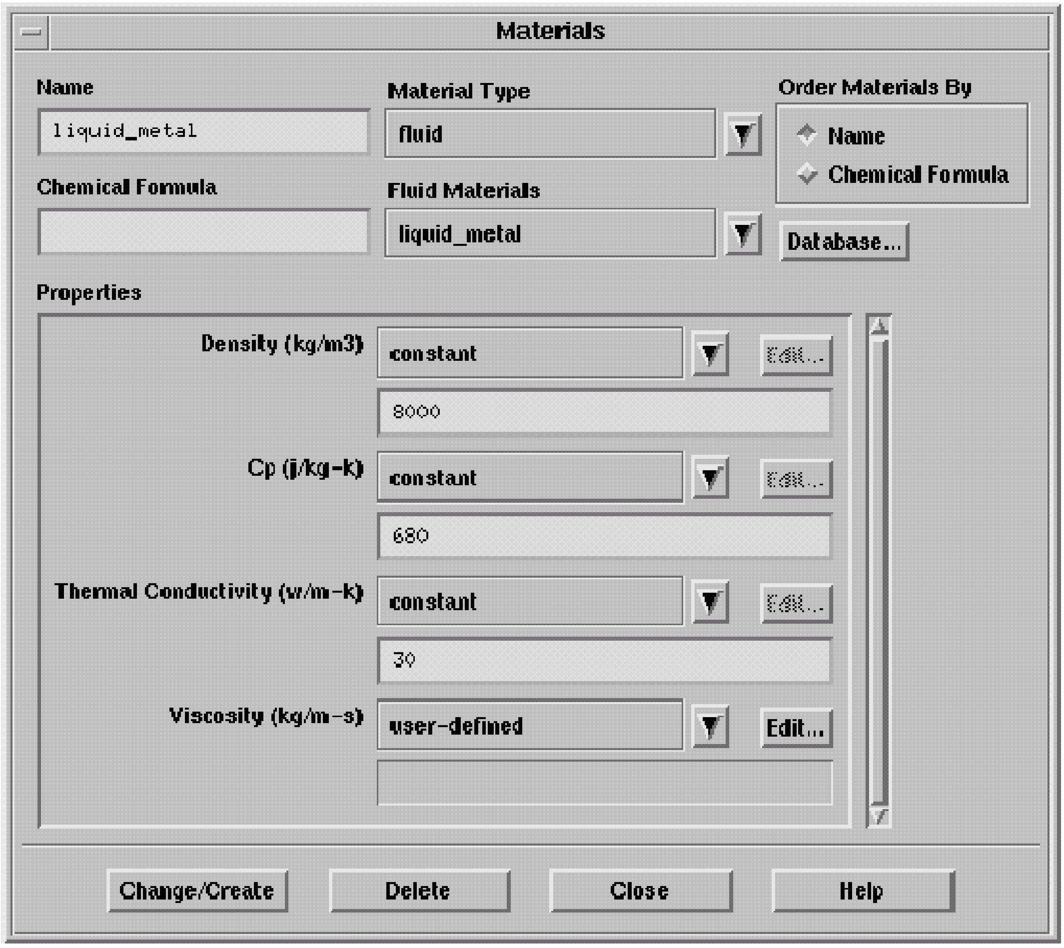


Figure 8.2.3: The Materials Panel

然后你需要在 **User-Defined Functions** 面板中选择希望的 UDF(如 cell_viscosity)(图 8.2.4)。



Figure 8.2.4: The User-Defined Functions Panel

！如果你计划使用一个 UDF 来定义密度，注意当密度变化增大时，求解收敛性将变得很差，指定一个可压缩定律（密度为压力的函数）或者多相行为（在空间变化的密度）可能会导致发散。建议你将 UDF 用于密度时限制在只有轻微密度变化的弱可压缩流动。

材料属性 UDF 用 **DEFINE_PROPERTY** 宏定义。细节见 4.3.6 节。对于用户定义标量或物质质量扩散率的 UDF 用 **DEFINE_DIFFUSIVITY** 宏定义。细节见 4.3.2 节。

8.2.5　预混燃烧源项

一旦你采用 7.2 节和 7.3 节中概括的方法之一编译（并连接）了你的预混燃烧源项 UDF，它将随之在 FLUENT 中是可见的和可选择的。你将需要在 **User-Defined Function Hooks** 面板中的 **Turbulent Premixed Source Function** 下拉列表中选择它。（图 8.2.5）

Define ⟶ User-Defined ⟶ Function Hooks...

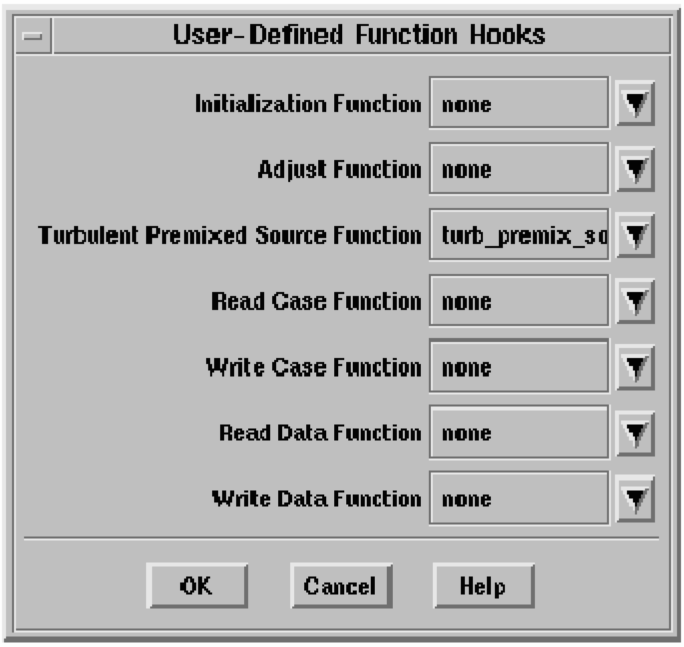


Figure 8.2.5: The User-Defined Function Hooks Panel

湍流预混速度和源项 UDF 用 **DEFINE_TURB_PREMIX_SOURCE** 宏定义。更多细节见 4.3.10。

8.2.6　反应速率

一旦你采用 7.2 节和 7.3 节中概括的方法之一编译（并连接）了你的反应速率 UDF，它将随之在 **FLUENT** 中是可见的和可选择的。你将需要在 **User-Defined Function Hooks** 面板中选择它。（图 8.1.1）

Define ⟶ User-Defined ⟶ Function Hooks...

你可以在 **Volume Reaction Rate Function** 或 **Surface Reaction Rate Function** 下拉列表中选择适当的 UDF。

表面和容积反应速率 **UDF** 用 **DEFINE_SR_RATE** 和 **DEFINE_VR_RATE** 宏定义。更多细节见 4.3.9 节和 4.3.14 节。

8.2.7　源项

一旦你采用 7.2 节和 7.3 节中概括的方法之一编译（并连接）了你的源项 UDF，它将随之在 **FLUENT** 中是可见的和可选择的。

你将需要在 **Fluid** 或 **Solid** 面板中打开 **Source Terms** 选项，并在适当的下拉列表里选择你的 UDF 的名字（如 cell_x_source）。（图 8.2.6）

对于源项的 UDF 用 **DEFINE_SOURCE** 宏定义。更多细节见 4.3.8 节。

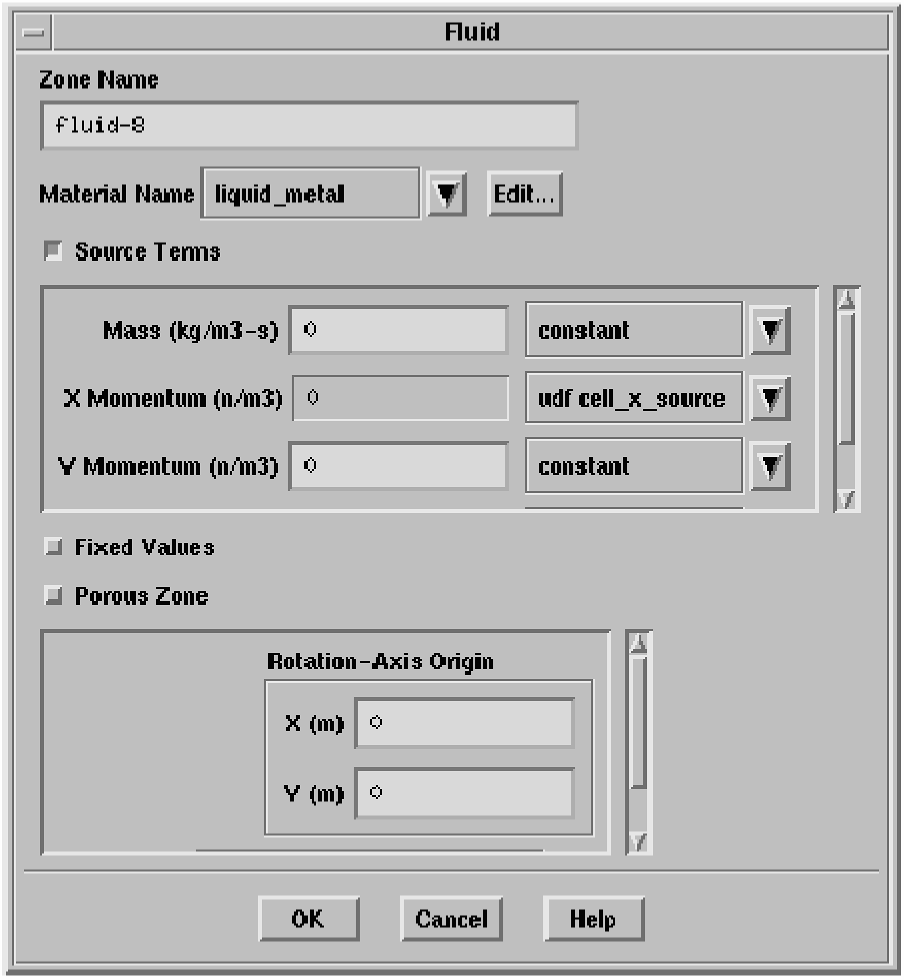


Figure 8.2.6: The Fluid Panel

8.2.8　时间步进

一旦你采用 7.2 节和 7.3 节中概括的方法之一编译（并连接）了你的用户时间步进 UDF，它将随之在 **FLUENT** 中是可见的和可选择的。你将首先需要在 **Iterate** 面板中选择时间步进方法 **Time Stepping Method** 为 **Adaptive**（图 8.2.7）。

Solve ⟶ Iterate...

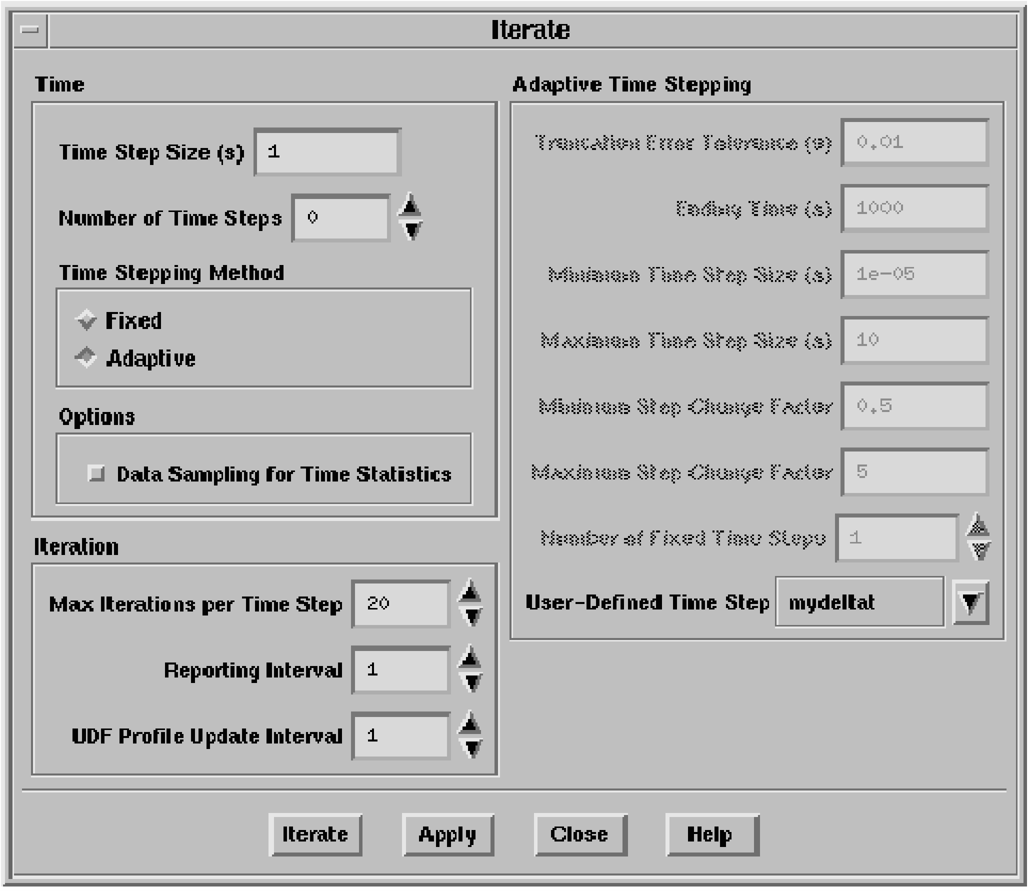


Figure 8.2.7: The Iterate Panel

接着，在 **Adaptive Time Stepping** 下的 **User_Defined Time Step** 下拉列表中选择你的 UDF 的名字（如 mydeltat）。

DEFINE_DELTAAT 宏用于在时间依赖计算中自定义时间步长。细节见 4.3.1 节。

8.2.9　湍流粘性

一旦你采用 7.2 节和 7.3 节中概括的方法之一编译（并连接）了你的湍流粘性 UDF 用于 Spalart-Allmaras、k-e、k-w 或 LES 湍流模型，它将随之在 FLUENT 中是可见的和可选择的。你将需要在 **Viscous Model** 面板中 **User-Defined Functions** 下的 **Turbulance Viscosity** 下拉列表中激活它（图 8.2.8）。

Define ⟶ Models ⟶ Viscous...

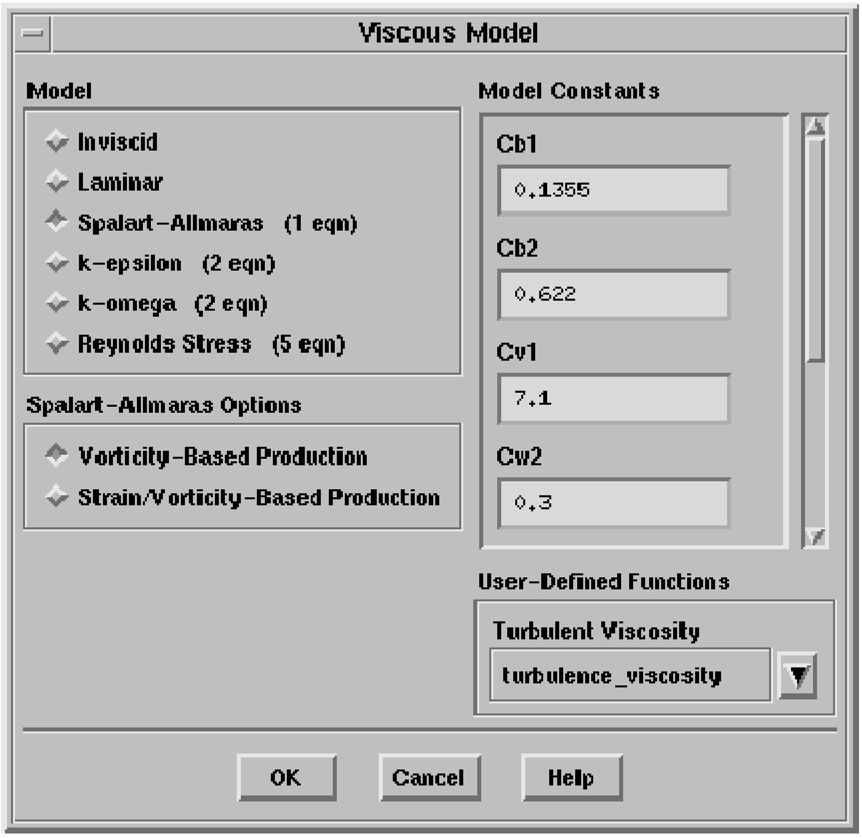


Figure 8.2.8: The Viscous Model Panel

对于湍流粘度的 UDF 用 **DEFINE_TURBULENT_VISCOSITY** 宏定义。更多细节见 4.3.11 节。

8.2.10 用户定义标量的通量

一旦你采用 7.2 节和 7.3 节中概括的方法之一编译（并连接）了你的 UDS 通量 UDF，它将随之在 **FLUENT** 中是可见的和可选择的。你将需要在 **User-Defined Scalars** 面板中激活它（图 8.2.9）。

Define → User-Defined → Scalars...

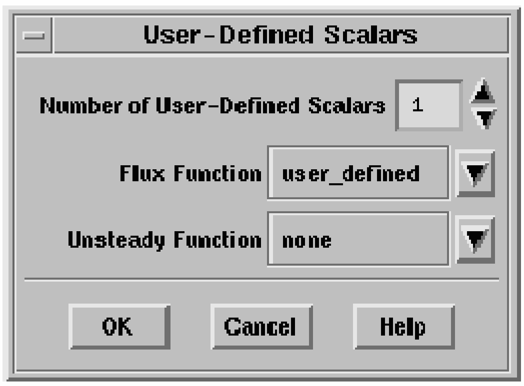


Figure 8.2.9: The User-Defined Scalars Panel

首先在 **User-Defined Scalars** 面板中指定 **Number of user-Defined Scales**，并且在 **Flux Functions** 下拉列表中选择适当的 UDF。

用户定义标量通量 UDF 用 **DEFINE_UDS_FLUX** 宏定义。更多细节见 4.3.12 节。

8.2.11 用户定义非稳态标量项

一旦你采用 7.2 节和 7.3 节中概括的方法之一编译（并连接）了你的非稳态 UDS 项 UDF，它将随之在 **FLUENT** 中是可见的和可选择的。你将需要在 **User-Defined Scalars** 面板中激活它（图 8.2.9）。

Define → User-Defined → Scalars...

首先指定 **Number of user-Defined Scales**，然后在 **Unsteady Function** 下拉列表中选择适当的 UDF。注意只有已经在 **Slover** 面板中指定了非稳态计算后，这一列表才会出现。

用户定义标量非稳态项 UDF 用 **DEFINE_UDS_UNSTEATY** 宏定义。更多细节见 4.3.12 节。

8.3 激活多相 UDF

本节包括激活使用 4.4 节中宏的 UDF 的方法。

8.3.1 气化速率

一旦你已经运用 7.2 节和 7.3 节中概括的方法之一编译（并连接）了你的气化速率 UDF，它将随之在 **FLUENT** 中成为可见的和可选择的。你将首先需要通过在 **Multiphase Model** 面板中选择 Cavitation 来使能相间质量输运。然后，在 **User-Defined Function Hooks** 面板中的 **Cavitation Mass Rate Function** 下拉列表中选择 UDF 的名字（图 8.1.1）。

Define → User-Defined → Function Hooks...

气化速率 UDF 以 **DEFINE_CAVITATION** 宏定义。更多细节见 4.4.1 节。

8.3.2 混合物模型的滑移速度

一旦你已经运用 7.2 节和 7.3 节中概括的方法之一编译（并连接）了你的自定义滑移速度 UDF，用于多相混合物模型，它将随之在 FLUENT 中成为可见的和可选择的。你将首先需要通过在 **Phase Interaction** 面板中 **Slip Velocity** 下的下拉列表里选择 user-defined(图 8.3.1)。

Define → Phases...

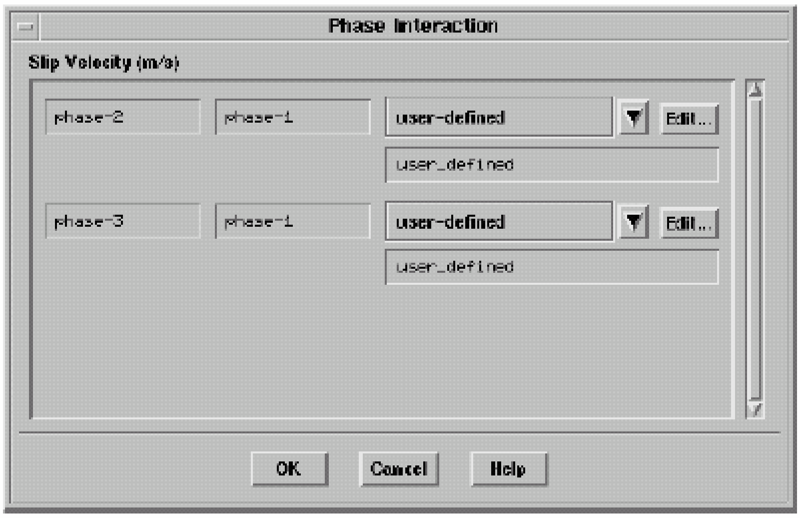


Figure 8.3.1: The Phase Interaction Panel

然后，在 **User-Defined Functions** 面板中选择希望的 UDF（如 slip_velocity）（图 8.3.2）。



Figure 8.3.2: The User-Defined Functions Panel

多相混合物模型的滑移速度 UDF 使用 **DEFINE_VECTOR_EXCHANGE_PROPERTY** 宏。更多细节见 4.4.3 节。

8.3.3　混合物模型的微粒直径

一旦你已经运用 7.2 节和 7.3 节中概括的方法之一编译（并连接）了你的微粒或液滴直径 UDF，用于多相混合物模型，它将随之在 FLUENT 中成为可见的和可选择的。你将首先需要通过在 **Secondary Phase** 面板中 **Diameter** 下拉列表里选择 user-defined(图 8.3.3)。然后，在 **User-Defined Function Hooks** 面板中的 **Cavitation Mass Rate Function** 下拉列表中选择 UDF 的名字（图 8.1.1）。

Define ⟶Phases...

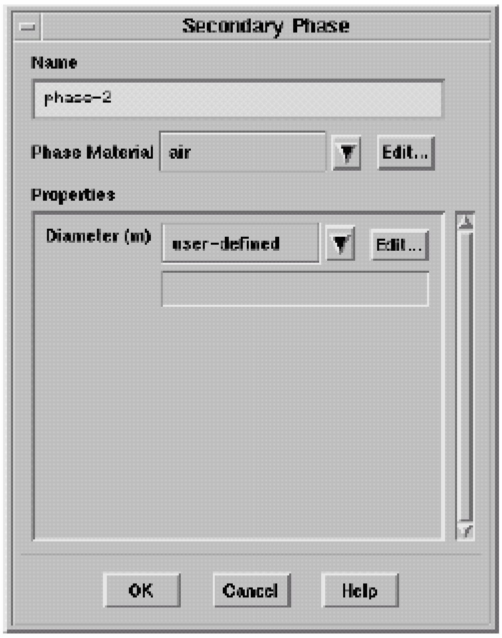


Figure 8.3.3: The Secondary Phase Panel

然后，在 **User-Defined Functions** 面板中选择希望的 UDF（如 diameter）（图 8.3.4）。



Figure 8.3.4: The User-Defined Functions Panel

DEFINE_PROPERTY 宏用于对微粒或液滴直径提供一种新定义。更多细节见 4.3.6 节。

8.3.4 欧拉模型的拖拉和提升系数

一旦你已经运用 7.2 节和 7.3 节中概括的方法之一编译（并连接）了你的拖拉和提升系数 UDF，用于欧拉多相模型，它将随之在 FLUENT 中成为可见的和可选择的。你将首先需要通过在 **Phase Interaction** 面板中 **Drag or Lift** 区域里的 **Drag Coefficient** 或 **Lift Coefficient** 下拉列表中选择 user-defined(图 8.3.5)。

Define ⟶ Phases...

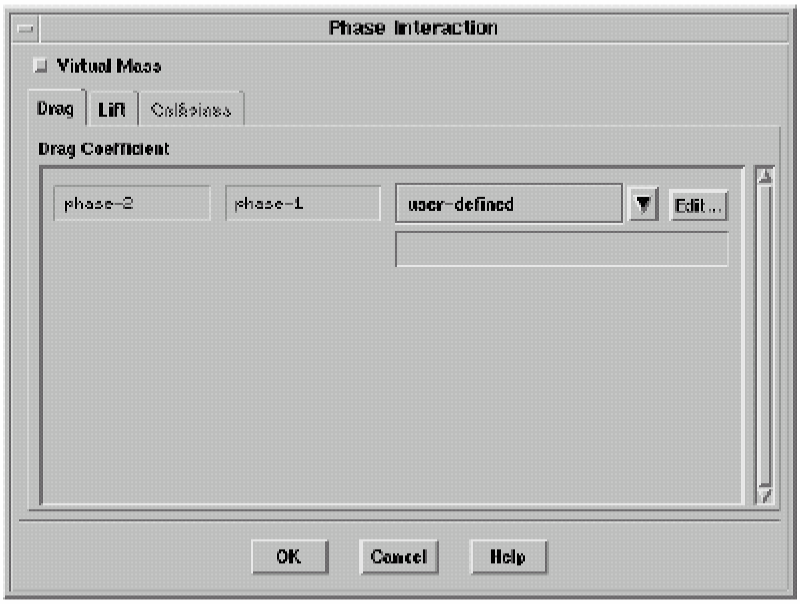


Figure 8.3.5: The Phase Interaction Panel

然后，在 **User-Defined Functions** 面板中选择希望的 UDF(图 8.3.4)。

用于多相欧拉模型的拖拉和提升系数 UDF 用 **DEFINE_EXCHANGE_PROPERTY** 宏定义。更多细节见 4.4.2 节。

8.4 激活 DPM UDF

本节包括激活中使用 4.5 节中宏的 UDF 的方法。

8.4.1 DPM 体积力

一旦你已经运用 7.2 节和 7.3 节中概括的方法之一编译（并连接）了你的离散相体积力 UDF，它将随之在 FLUENT 中成为可见的和可选择的。你将需要在 **Discrete Phase Model** 面板中 **User-Defined Function** 下的 **Body Force** 下拉列表中选择 UDF 的名字（图 8.4.1）。

Define → Models → Discrete Phase...

用于 DPM 的体积力 UDF 以 **DEFINE_DPM_BODY_FORCE** 宏定义。更多细节见 4.5.1 节。

8.4.2 DPM 的拖拉系数

一旦你已经运用 7.2 节和 7.3 节中概括的方法之一编译（并连接）了你的离散相拖拉系数 UDF，它将随之在 FLUENT 中成为可见的和可选择的。你将需要在 **Discrete Phase Model** 面板中 **Drag Parameters** 下的 **Drag Law** 下拉列表中选择 UDF 的名字（图 8.4.1）。

Define → Models → Discrete Phase...

DPM 的拖拉系数 UDF 以 **DEFINE_DPM_BODY_DRAG** 宏定义。更多细节见 4.5.2 节。

8.4.3 DPM 的腐蚀和增长速率

一旦你已经运用 7.2 节和 7.3 节中概括的方法之一编译（并连接）了你的离散相 DPM 的腐蚀和增长速率 UDF，它将随之在 FLUENT 中成为可见的和可选择的。你将需要在 **Discrete Phase Model** 面板中 **User-Defined Function** 下的 **Erosion/Accretion** 下拉列表中选择 UDF 的名字（图 8.4.1）。

Define → Models → Discrete Phase...

DPM 的腐蚀和增长速率 UDF 以 **DEFINE_DPM_EROSION** 宏定义。更多细节见 4.5.3 节。

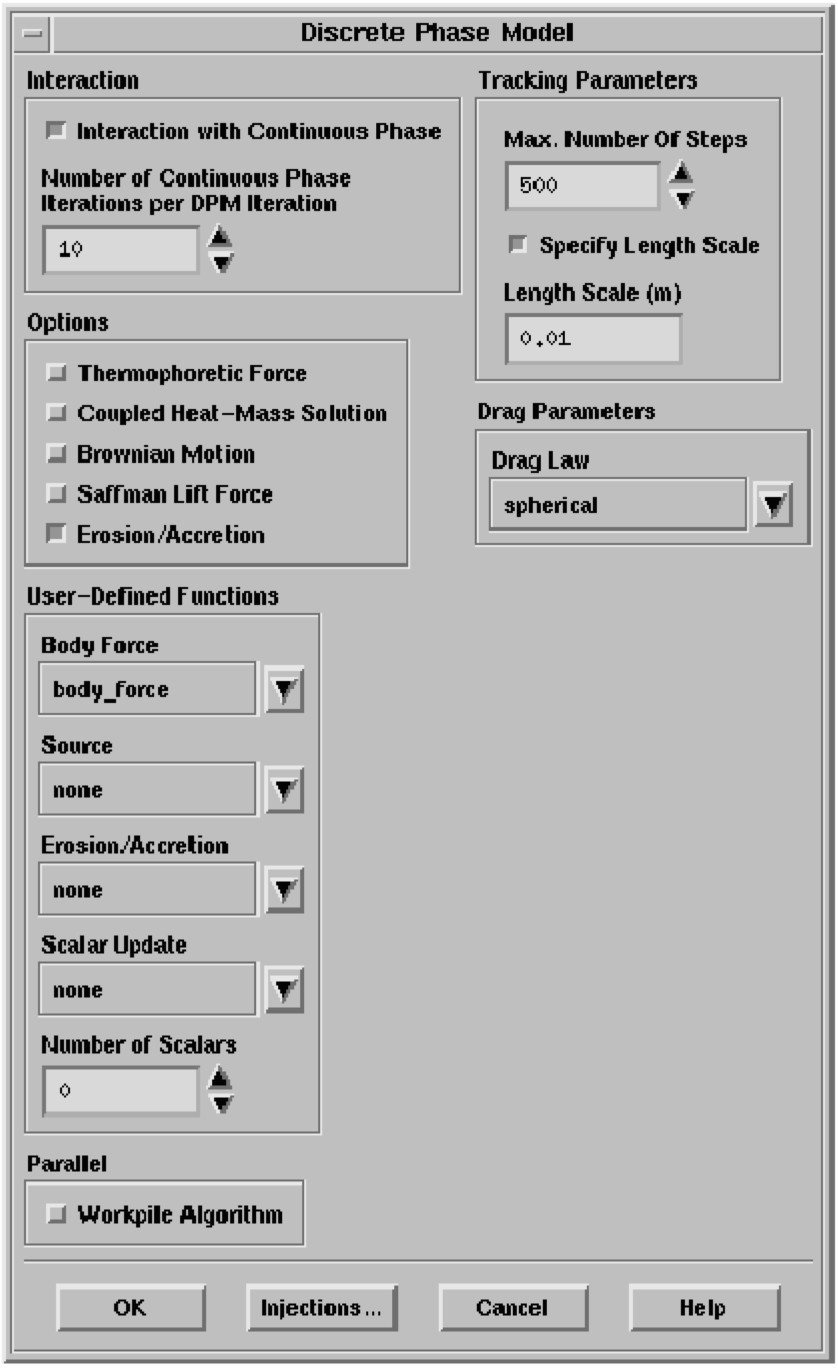


Figure 8.4.1: The Discrete Phase Model Panel

8.4.3 DPM 初始化

一旦你已经运用 7.2 节和 7.3 节中概括的方法之一编译（并连接）了你的离散相初始化 UDF，它将随之在 **FLUENT** 中成为可见的和可选择的。你将需要在 **Set Injection Properties** 面板中的 UDF 区域内，**User-Defined Functions** 下的 **Initialization** 下拉列表中选择 UDF 的

名字（图 8.4.2）

Define ⟶ Injections...

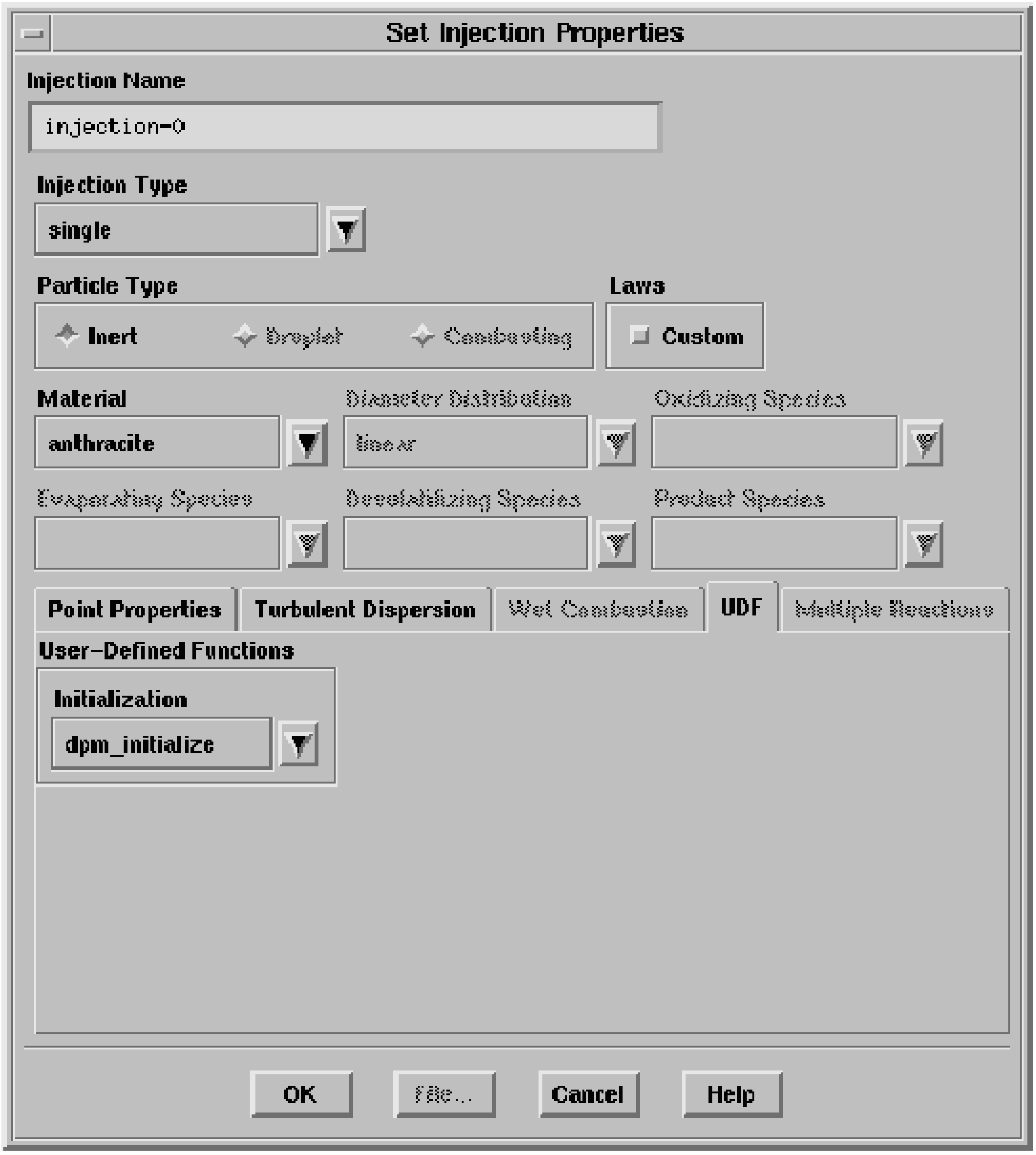


Figure 8.4.2: The Set Injection Properties Panel

DPM 的初始化 UDF 以 **DEFINE_DPM_INJECTION_INIT** 宏定义。更多细节见 4.5.4 节。

8.4.5　用户 DPM 定律

一旦你已经运用 7.2 节和 7.3 节中概括的方法之一编译（并连接）了你的离散相用户定律或转换 UDF，它将随之在 **FLUENT** 中成为可见的和可选择的。你将需要在 Custom Laws 面板中的适当下拉列表里选择 UDF 的名字（图 8.4.3）。为打开 Custom Laws 面板，需要使能 **Set Injection Properties** 面板中 **Laws** 下的 **Custom** 选项。

Define ⟶ Injections...



Figure 8.4.3: The Custom Laws Panel

在六种微粒定律左边的下拉列表里，你都可以针对用户定律选择适当的微粒定律 UDF。第 7 个下拉列表标记为 Switching，能用于改变使用的用户定律。你可以通过在这一下拉列表中选择一个 UDF 来定制 FLUENT 在定律之间转换的方式。

DPM 的用户定律 UDF 用 **DEFINE_DPM_LAW** 宏定义。你可以使用 **DEFINE_DPM_SWITCH** 宏来修改定律之间转换的标准。更多细节见 4.5.5 节和 4.5.10 节。

8.4.5 DPM 输出

一旦你已经运用 7.2 节和 7.3 节中概括的方法之一编译(并连接)了你的离散相输出 UDF，它将随之在 FLUENT 中成为可见的和可选择的。你将需要在 **Sample Trajectories** 面板中 **User-Defined Functions** 下的 **Output** 下拉列表中选择这一 UDF 的名字（图 8.4.4）。

Report ⟶ Discrete Phase ⟶ Sample...

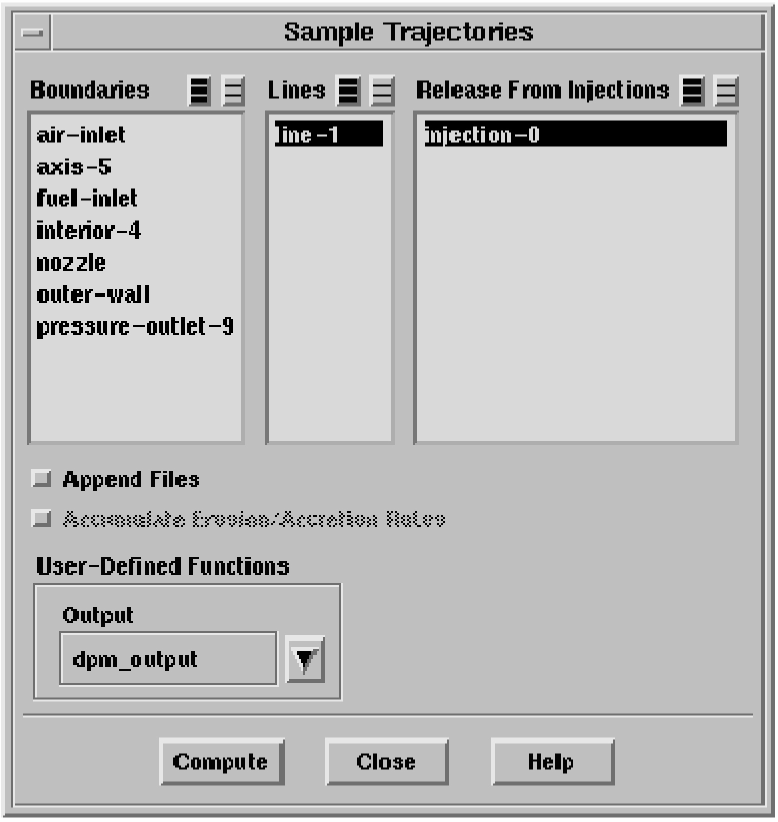


Figure 8.4.4: The Sample Trajectories Panel

DPM 的输出 UDF 用 **DEFINE_DPM_OUTPUT** 宏定义。更多细节见 4.5.6 节。

8.4.5　DPM 材料属性

一旦你已经运用 7.2 节和 7.3 节中概括的方法之一编译(并连接)了你的离散相属性 UDF，它将随之在 **FLUENT** 中成为可见的和可选择的。你将需要在 **Materials** 面板中适当的属性的下拉列表中选择这一 UDF 的名字（图 8.2.3)。

Define ⟶ Materials...

然后，在 **User-Defined Functions** 面板中选择希望的 UDF。

DPM 的属性 UDF 用 **DEFINE_DPM_PROPERTY** 宏定义。更多细节见 4.5.7 节。

8.4.8　DPM 标量更新

一旦你已经运用 7.2 节和 7.3 节中概括的方法之一编译（并连接）了你的离散相标量更新 UDF，它将随之在 FLUENT 中成为可见的和可选择的。你将需要在 **Discrete Phase Model** 面板中 **User-Defined Functions** 下的 **Scalar Update** 下拉列表中选择这一 UDF 的名字（图 8.4.1）你还需要指定 **Number of Scalars**。

Define ⟶ Models ⟶ Discrete Phase...

DPM 标量更新 UDF 用 **DEFINE_DPM_SCALAR_UPDATE** 宏定义。更多细节见 4.5.8 节。

8.4.8　DPM 源项

一旦你已经运用 7.2 节和 7.3 节中概括的方法之一编译(并连接)了你的离散相源项 UDF，

它将随之在 **FLUENT** 中成为可见的和可选择的。

你将需要在 **Fluid** 面板中打开 **Source Terms** 选项，并在适当的下拉列表中选择这一 UDF 的名字（如 cell_x_source）(图 8.2.6)。

DPM 的源项 UDF 用 **DEFINE_DPM_SOURCE** 宏定义。更多细节见 4.5.9 节。

第九章

本章扼要介绍了 FLUENT 中用户自定义标量及它们的用法。

- 9.1 介绍
- 9.2 理论
- 9.3 UDS 的定义，求解，后处理

9.1 介绍

FLUENT 可以用求解诸如质量组分之类标量方程的相同方法来求解任意的用户自定义标量 (UDS)。在某些类型的应用中，如燃烧模拟或是等离子增强表面反应（plasma-enhanced surface reaction）的模拟中，还需引入新的标量输运方程。用户自定义标量可被用于磁流体动力(MHD)模拟中。在 MHD 中，导电流体（conducting fluid）的流体将会产生磁场，此磁场可以用户自定义标量来求解。磁场造成的对流体的阻尼（a resistance to the flow），可用用户自定义的源项来模拟。书中 4.3.12 和 4.3.13 介绍了用 UDFs 来定义标量输运方程的例子。 to customize scalar transport equations.

9.2 理论

对于一个任意的标量 ϕ_i， FLUENT 可求解方程

$$\frac{\partial \rho \phi_i}{\partial t} + -\nabla \cdot (\Gamma_i \nabla \phi_i) = S_{\phi_i} \quad i = 1, ..., N \tag{9.2.1}$$

此处 Γ_i 和 S_{ϕ_i} 是用户为 N 个标量方程中的每一个方程定义的扩散系数和源项。对于稳态的情况，根据计算对流通量的方法的不同，FLUENT 可求解以下的三种方程之一：

- 如果对流通量不用计算，则 FLUENT 可解方程

$$-\nabla \cdot (\Gamma_i \nabla \phi_i) = S_{\phi_i} \quad i = 1, \ldots, N \qquad (9.2.2)$$

此处 Γ_i 和 S_{ϕ_i} 是用户为 N 个标量方程中的每一个方程定义的扩散系数和源项。

- 如果以质量流率来计算对流通量，FLUENT 可解方程

$$\nabla \cdot (\rho \vec{v} \phi_i - \Gamma_i \nabla \phi_i) = S_{\phi_i} \quad i = 1, \ldots, N \qquad (9.2.3)$$

- 如果选择一个用户自定义函数来计算对流通量，FLUENT 可解方程

$$\nabla \cdot (\vec{J} \phi_i - \Gamma_i \nabla \phi_i) = S_{\phi_i} \quad i = 1, \ldots, N \qquad (9.2.4)$$

此处 $\vec{J}$ 是用户定义的流率。

!! 在 FLUENT 中，用户自定义函数只可在流体区域内求解，而不能在固体区域内求解。

9.3 UDS 的定义，求解，后处理

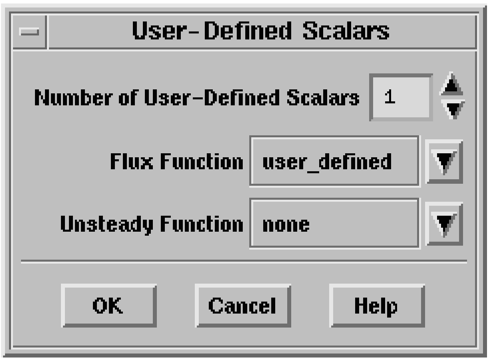
定义，求解，后处理用户自定义标量的步骤概括如下。注意 UDFs 在多相流体和单项流体中应用的重要不同在于，如果是单相的情况（an individual phase）， 用户需要提供用户自定义的标量通量函数。这是因为缺省的用户自定义标量通量函数是在混合物区域（the mixture domain）上定义的，如果用户在单相流体中使用它将会导致质量不平衡。用户需要确保自定义的通量函数 UDF 可提供给求解器适当的各相值（phase values）。应用于用户自定义标量(UDS)的 DEFINE 宏见 4.3.12 和 4.3.13 。

1. 在 **User-Defined Scalars** panel (Figure9.3.1)中选择标量数目

Define →User-Defined →Scalars...

!! 用户自定义的标量输运方程数目最大为 50。

Figure 9.3.1: The **User-Defined Scalars** Panel



2. 选项 **Flux Function** 有 **none**, **mass flow rate**, 或 a user-defined function。用户自定义标量通量函数（User-defined scalar flux functions）用 DEFINE_UDS_FLUX 宏 (见 4.3.12)来定义。所有已被定义的用户自

定义函数将会出现在 **Flux Function** 列表中。通量函数决定了对流通量怎样计算，以及 **FLUENT** 求解哪一类的 UDS 方程。选择 **none, mass flow rate**, 或是用户自定义函数将会使 **FLUENT** 分别求解方程 9.2-2，9.2-3 或 9.2-4。

> !! 用户选定 Flux Function 将适用于所有 UDS 的 Flux Function 。如果用户有多个 UDS 的 Flux Function, 所有对流通量的计算将以同一方式进行 。如果用户选择的是用户自定义函数，计算将包括所有的 UDS 的通量函数。

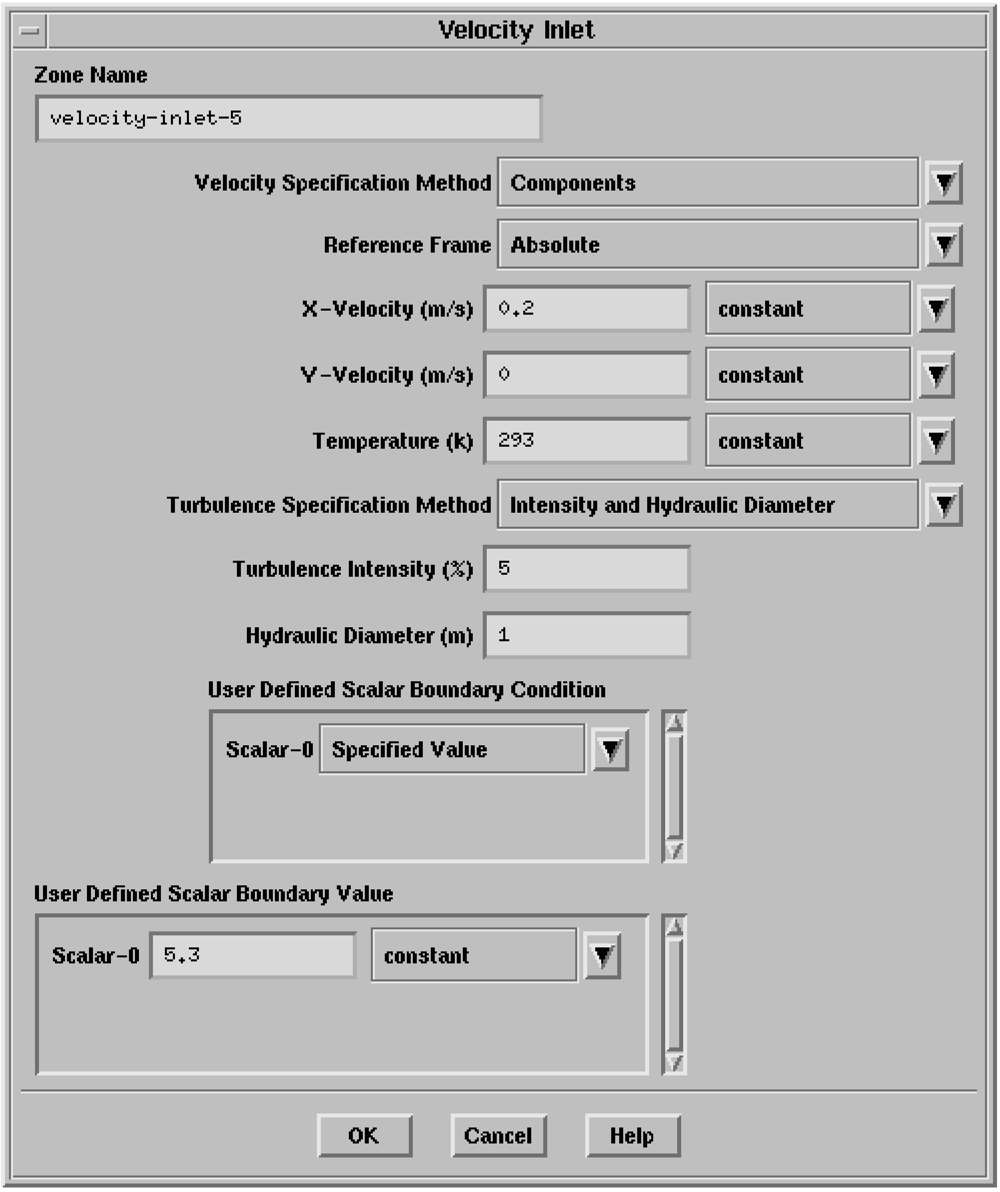
3. 选定 **Unsteady Function** 为 **none, default**,或是用户自定义函数 (所有已被定义的用户自定义函数将会出现在 **Unsteady Function** 列表中) 。选择 **none** 为稳定状态的求解，如果用户需要求解方程中的时间项 9.2-1 则需选择 **default**, 选择 **user-defined** 则可使用户定义的 UDF 调用 DEFINE_UDS_UNSTEADY 宏。详情请见 4.3.13。

4. 为 UDS 选定在所有壁面上，入口处，出口处的边界条件。用户可为每个标量定义一个特定值或是特定的通量。

 Define →Boundary Conditions...

 (a) 在 **User Defined Scalar Boundary Condition** 下 (如 Figure9.3.2), 在与每个标量相邻的下拉列表中选择 **Specified Flux** 或是 **Specified Value**。

Figure 9.3.2: The **Velocity Inlet** Panel with Inputs for a User-Defined Scalar



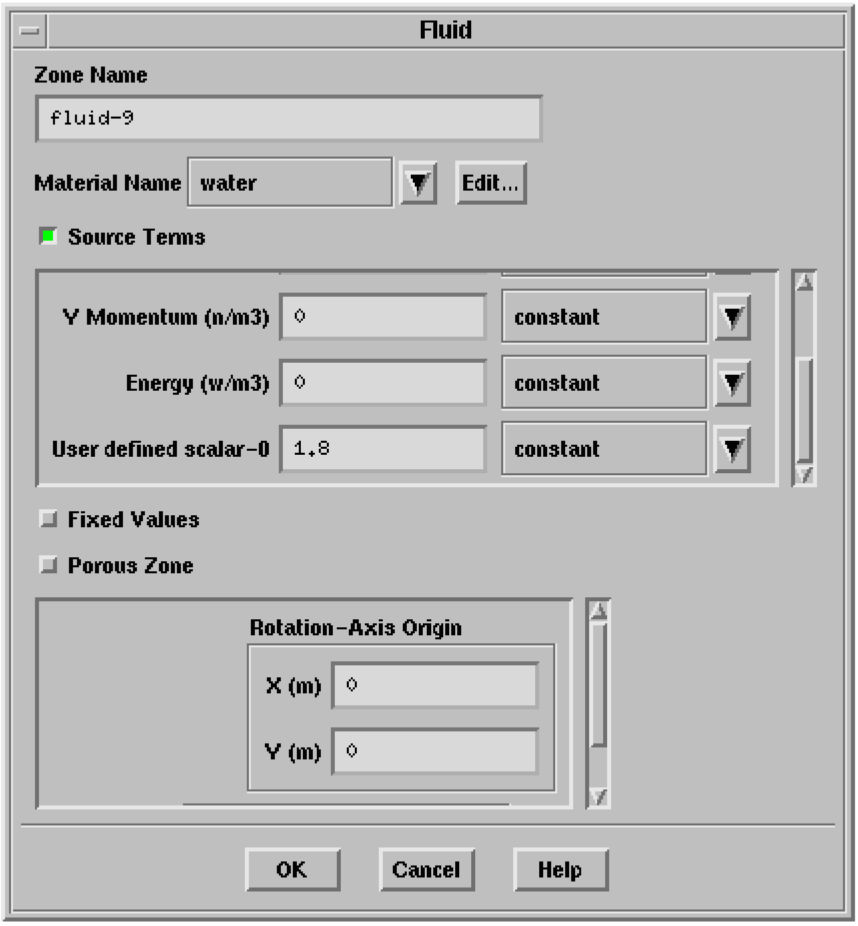
(b) 在 **User Defined Scalar Boundary Value** 中，为每个用户自定义函数输入一个常数值或是一个用户自定义函数。如果用户选择了 **Specified Flux**,用户输入的将是边界的通量值(例如方程 9.2-2 左边括号中的负项点乘 **n** ，此处 **n**垂直于该边界。如果用

户选择 **Specified Value**，用户输入的则是标量本身在边界的值。

5. 如果 UDS 方程中含有源项，在需激活 **Fluid** 面板里的 **Source Terms** 选项，且将此标量的源项设置为一常数值或是用户自定义函数（见 Figure9.3.3）。如果用户选择源项为用户自定义函数，用户函数必须计算源项 S 和它的导数 $\partial S/\partial \phi$。

Define →Boundary Conditions...

Figure 9.3.3: The **Fluid** Panel with Inputs for Source Terms for a User-Defined Scalar



6. 设置方案参数（solution parameters），为每个 UDS 指定初始值（如

同别的标量输运方程)，然后计算。

7. 使用普通的后处理工具来检查结果。在每个后处理面板中，场变量列表中都将包含 **User Defined Scalars...** ，其中包含每个 UDS 的值和它的扩散系数 (方程 9.2-1，9.2-2，9.2-3 或 9.2-4 中的 Γ_i):

- Scalar-n
- **Diffusion Coef. of Scalar-n**

10.5 提供了一些用户自定义标量输运方程的例子。

第九章

本章扼要介绍了 FLUENT 中用户自定义标量及它们的用法。

- 9.1 介绍
- 9.2 理论
- 9.3 UDS 的定义，求解，后处理

9.1 介绍

FLUENT 可以用求解诸如质量组分之类标量方程的相同方法来求解任意的用户自定义标量 (UDS)。在某些类型的应用中，如燃烧模拟或是等离子增强表面反应（plasma-enhanced surface reaction）的模拟中，还需引入新的标量输运方程。用户自定义标量可被用于磁流体动力(MHD)模拟中。在 MHD 中，导电流体（conducting fluid）的流体

将会产生磁场，此磁场可以用户自定义标量来求解。磁场造成的对流体的阻尼（a resistance to the flow），可用用户自定义的源项来模拟。书中 4.3.12 和 4.3.13 介绍了用 UDFs 来定义标量输运方程的例子。 to customize scalar transport equations.

9.2 理论

对于一个任意的标量 ϕ_i， FLUENT 可求解方程

$$\frac{\partial \rho \phi_i}{\partial t} + -\nabla \cdot (\Gamma_i \nabla \phi_i) = S_{\phi_i} \quad i = 1, ..., N \tag{9.2.1}$$

此处 Γ_i 和 S_{ϕ_i} 是用户为 N 个标量方程中的每一个方程定义的扩散系数和源项。对于稳态的情况，根据计算对流通量的方法的不同，FLUENT 可求解以下的三种方程之一：

- 如果对流通量不用计算，则 FLUENT 可解方程

$$-\nabla \cdot (\Gamma_i \nabla \phi_i) = S_{\phi_i} \quad i = 1, ..., N \tag{9.2.2}$$

  此处 Γ_i 和 S_{ϕ_i} 是用户为 N 个标量方程中的每一个方程定义的扩散系数和源项。

- 如果以质量流率来计算对流通量，FLUENT 可解方程

$$\nabla \cdot (\rho \vec{v} \phi_i - \Gamma_i \nabla \phi_i) = S_{\phi_i} \quad i = 1, ..., N \tag{9.2.3}$$

- 如果选择一个用户自定义函数来计算对流通量，FLUENT 可解方程

$$\nabla \cdot (\vec{J}\phi_i - \Gamma_i \nabla \phi_i) = S_{\phi_i} \quad i = 1, ..., N \qquad (9.2.4)$$

此处 $\vec{J}$是用户定义的流率。

!! 在 FLUENT 中，用户自定义函数只可在流体区域内求解，而不能在固体区域内求解。

9.3 UDS 的定义，求解，后处理

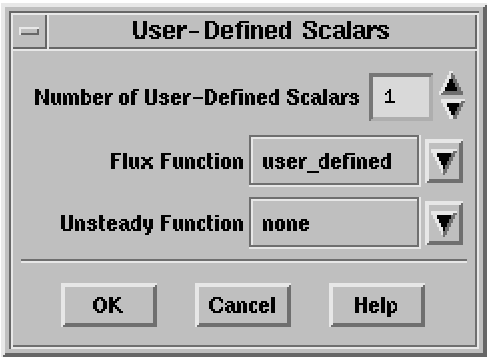
定义，求解，后处理用户自定义标量的步骤概括如下。注意 UDFs 在多相流体和单项流体中应用的重要不同在于，如果是单相的情况（an individual phase）， 用户需要提供用户自定义的标量通量函数。这是因为缺省的用户自定义标量通量函数是在混合物区域（the mixture domain）上定义的，如果用户在单相流体中使用它将会导致质量不平衡。用户需要确保自定义的通量函数 UDF 可提供给求解器适当的各相值（phase values）。应用于用户自定义标量(UDS)的 DEFINE 宏见 4.3.12 和 4.3.13 。

1. 在 **User-Defined Scalars** panel (Figure9.3.1)中选择标量数目

   Define →User-Defined →Scalars...

   !! 用户自定义的标量输运方程数目最大为 50。

Figure 9.3.1: The **User-Defined Scalars** Panel



2. 选项 **Flux Function** 有 **none**, **mass flow rate**, 或 a user-defined function。用户自定义标量通量函数（User-defined scalar flux functions）用 DEFINE_UDS_FLUX 宏 (见 4.3.12)来定义。所有已被定义的用户自定义函数将会出现在 **Flux Function** 列表中。通量函数决定了对流通量怎样计算，以及 **FLUENT** 求解哪一类的 UDS 方程。选择 **none**, **mass flow rate**, 或是用户自定义函数将会使 **FLUENT** 分别求解方程 9.2-2，9.2-3 或 9.2-4。

!! 用户选定 Flux Function 将适用于所有 UDS 的 Flux Function 。如果用户有多个 UDS 的 Flux Function, 所有对流通量的计算将以同一方式进行 。如果用户选择的是用户自定义函数，计算将包括所有的 UDS 的通量函数。

3. 选定 **Unsteady Function** 为 **none**, **default**,或是用户自定义函数 (所有已被定义的用户自定义函数将会出现在 **Unsteady Function** 列表中) 。选择 **none** 为稳定状态的求解，如果用户需要求解方程中的时

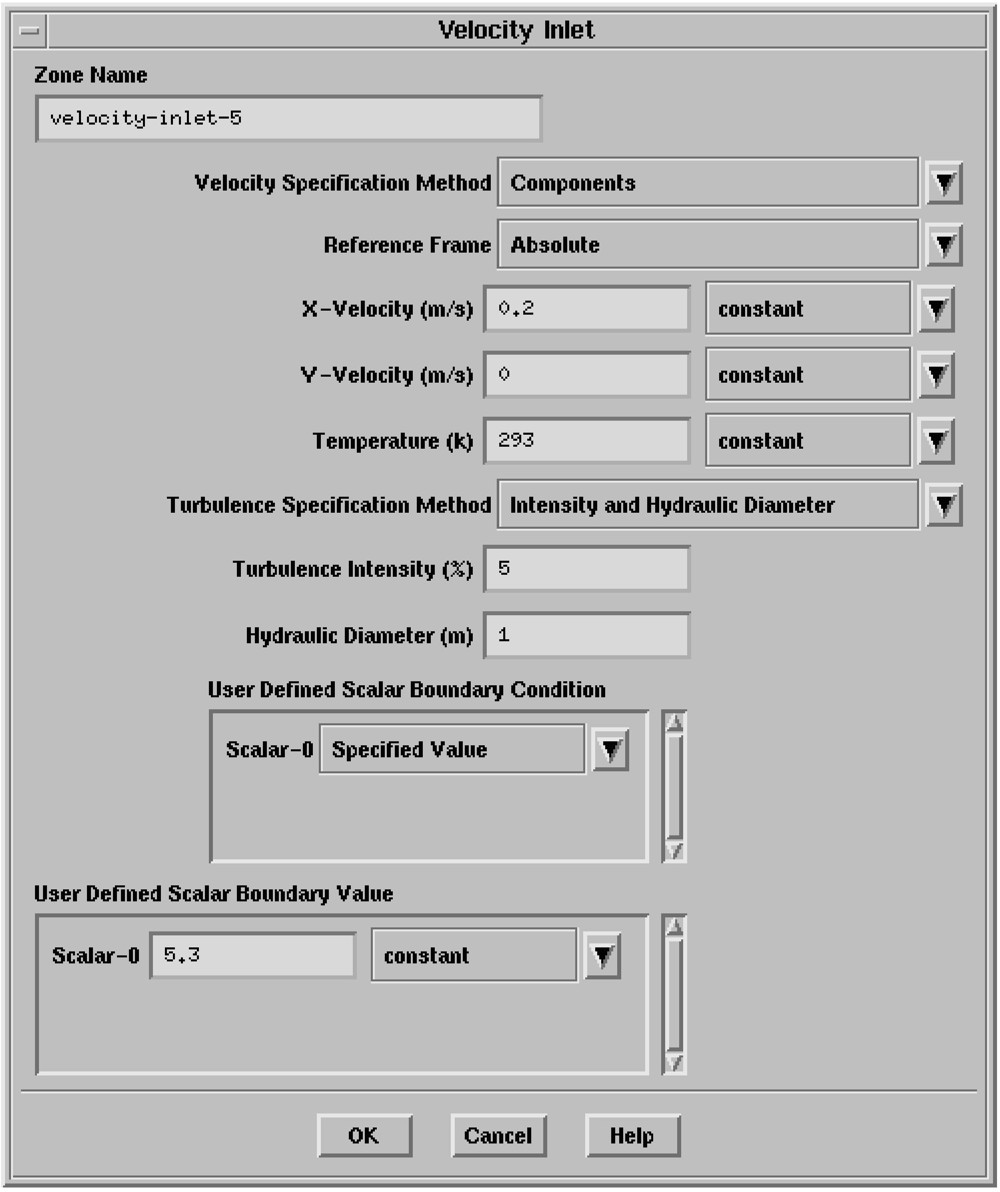
间项 9.2-1 则需选择 **default**, 选择 **user-defined** 则可使用用户定义的 UDF 调用 DEFINE_UDS_UNSTEADY 宏。详情请见 4.3.13。

4. 为 UDS 选定在所有壁面上，入口处，出口处的边界条件。用户可为每个标量定义一个特定值或是特定的通量。

Define →Boundary Conditions...

(a) 在 **User Defined Scalar Boundary Condition** 下 (如 Figure9.3.2), 在与每个标量相邻的下拉列表中选择 **Specified Flux** 或是 **Specified Value**。

Figure 9.3.2: The **Velocity Inlet** Panel with Inputs for a User-Defined Scalar



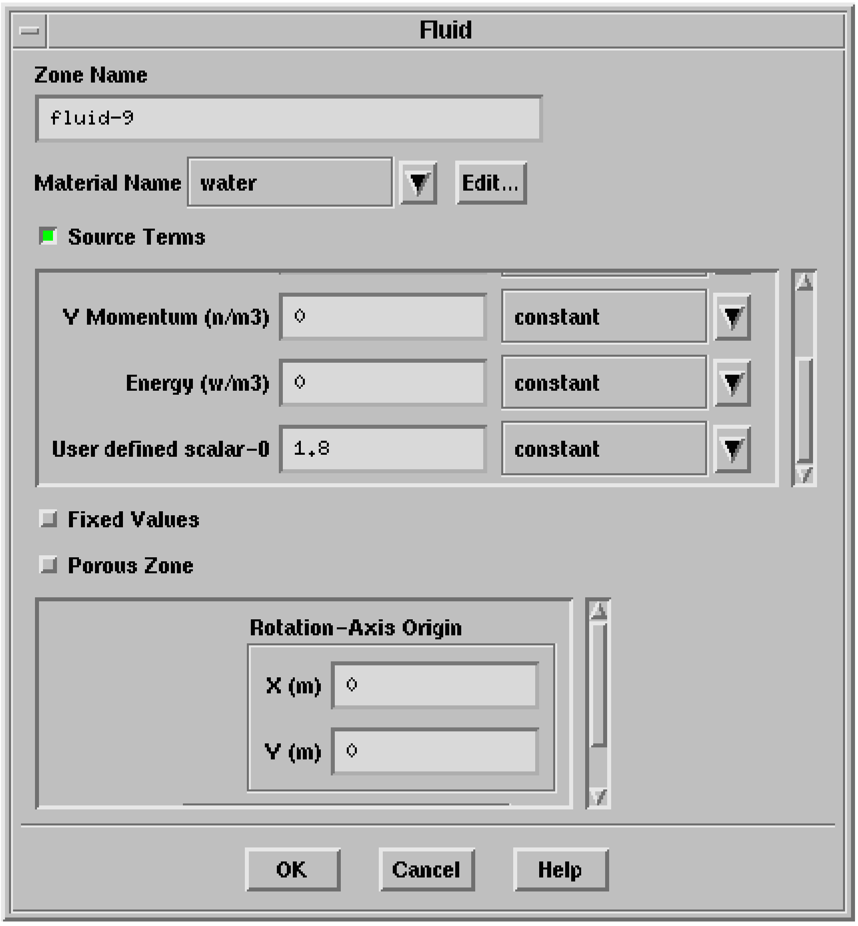
(b) 在 **User Defined Scalar Boundary Value** 中，为每个用户自定义函数输入一个常数值或是一个用户自定义函数。如果用户选择了 **Specified Flux**,用户输入的将是边界的通量值(例如方程 9.2-2 左边括号中的负项点乘 **n** ，此处 **n** 垂直于该边界。如果用户选择 **Specified Value**，用户输入的则是标量本身在边界的值。

5. 如果 UDS 方程中含有源项，在需激活 **Fluid** 面板里的 **Source**

Terms 选项，且将此标量的源项设置为一常数值或是用户自定义函数（见 Figure9.3.3）。如果用户选择源项为用户自定义函数，用户函数必须计算源项 S 和它的导数 $\partial S/\partial \phi$ 。

Define →Boundary Conditions...

Figure 9.3.3: The **Fluid** Panel with Inputs for Source Terms for a User-Defined Scalar



6. 设置方案参数（solution parameters），为每个 UDS 指定初始值（如同别的标量输运方程），然后计算。

7. 使用普通的后处理工具来检查结果。在每个后处理面板中，场变

量列表中都将包含 **User Defined Scalars...** ，其中包含每个 UDS 的值和它的扩散系数 (方程 9.2-1，9.2-2，9.2-3 或 9.2-4 中的 Γ_i):

- Scalar-n
- **Diffusion Coef. of Scalar-n**

10.5 提供了一些用户自定义标量输运方程的例子。